

Михаил Фленов

Библия C#

6-е издание

Программирование для .NET на C#
Платформа .NET 7
Базы данных
Entity Framework
Веб-программирование
Повторное использование кода
Изучение языка на полезных примерах



Материалы
на www.bhv.ru



Михаил Фленов

Библия

С#

6-е издание

Санкт-Петербург

«БХВ-Петербург»

2024

УДК 004.438 С#
ББК 32.973.26-018.1
Ф71

Фленов М. Е.

Ф71 Библия С#. — 6-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2024. — 512 с.: ил.

ISBN 978-5-9775-1828-4

Книга посвящена программированию на языке С# для платформы Microsoft .NET, начиная с основ языка и разработки программ для работы в режиме командной строки и заканчивая созданием современных веб-приложений. Материал сопровождается большим количеством практических примеров. Подробно описывается логика выполнения каждого участка программы. Уделено внимание вопросам повторного использования кода. В 6-м издании примеры переписаны с учетом современной платформы .NET 7, большой упор сделан на веб-приложения и фреймворк доступа к данным Entity Framework. На сайте издательства находятся коды программ, дополнительная справочная информация и копия базы данных для выполнения примеров из книги.

Для программистов

УДК 004.438 С#
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Людмила Гауль</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Дизайн обложки	<i>Инны Тачиной</i>
Оформление обложки	<i>Зои Канторович</i>

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20

Оглавление

Предисловие	9
Благодарности.....	13
Бонус	15
Структура книги	17
Глава 1. Введение в .NET.....	19
1.1. Платформа .NET	19
1.1.1. «Кубики» .NET	20
1.1.2. Сборки.....	21
1.2. Обзор среды разработки Visual Studio .NET	23
1.2.1. Работа с проектами и решениями	23
1.2.2. Работа с файлами	31
1.3. Простейший пример .NET-приложения	31
1.3.1. Проект на языке C#.....	31
1.3.2. Компиляция и запуск проекта на языке C#.....	32
1.4. Компиляция приложений.....	34
1.4.1. Компиляция в .NET Framework.....	34
1.4.2. Компиляция в .NET Core или .NET 5 и старше	35
1.5. Поставка сборок.....	37
1.6. Формат исполняемого файла .NET	40
1.7. Код верхнего уровня.....	41
Глава 2. Основы C#.....	42
2.1. Комментарии.....	42
2.2. Переменная	43
2.3. Именованние элементов кода	46
2.4. Работа с переменными	49
2.4.1. Строки и символы	52
2.4.2. Массивы	55
2.4.3. Перечисления	58
2.5. Простейшая математика.....	61

2.6. Условные операторы	66
2.6.1. Условный оператор <i>if</i>	66
2.6.2. Условный оператор <i>switch</i>	69
2.6.3. Сокращенная проверка	70
2.7. Циклы	71
2.7.1. Цикл <i>for</i>	71
2.7.2. Цикл <i>while</i>	74
2.7.3. Цикл <i>do...while</i>	75
2.7.4. Цикл <i>foreach</i>	76
2.8. Управление циклом	77
2.8.1. Оператор <i>break</i>	77
2.8.2. Оператор <i>continue</i>	78
2.9. Константы	79
2.10. Нулевые значения	80
2.11. Начальные значения переменных	80
2.12. Системы счисления	81
2.13. Логические операции	84

Глава 3. Объектно-ориентированное программирование 86

3.1. Объекты на C#	86
3.2. Свойства	90
3.3. Методы	96
3.3.1. Описание методов	97
3.3.2. Параметры методов	99
3.3.3. Перегрузка методов	107
3.3.4. Конструктор.....	107
3.3.5. Статичность	112
3.3.6. Рекурсия.....	115
3.3.7. Деструктор.....	118
3.3.8. Упрощенный синтаксис.....	119
3.4. Метод <i>Main()</i>	120
3.5. Оператор <i>Using</i>	122
3.6. Объекты только для чтения	123
3.7. Объектно-ориентированное программирование.....	124
3.7.1. Наследование.....	124
3.7.2. Инкапсуляция	128
3.7.3. Полиморфизм	131
3.8. Наследование от класса <i>Object</i>	133
3.9. Переопределение методов	134
3.10. Обращение к предку из класса	138
3.11. Вложенные классы	140
3.12. Область видимости	142
3.13. Ссылочные и простые типы данных	144
3.14. Проверка класса объекта.....	145
3.15. Неявный тип данных <i>var</i>	146
3.16. Абстрактные классы.....	147
3.17. Инициализация свойств	150
3.18. Частицы класса	151

Глава 4. Консольные приложения.....	152
4.1. Украшение консоли.....	153
4.2. Работа с буфером консоли	155
4.3. Окно консоли	157
4.4. Запись в консоль	157
4.5. Чтение данных из консоли.....	160
Глава 5. Продвинутое программирование.....	162
5.1. Приведение и преобразование типов	162
5.2. Все в .NET — это объекты.....	164
5.3. Работа с перечислениями <i>Enum</i>	166
5.3.1. Перечисления — это значения	166
5.3.2. Перечисления-флаги	169
5.4. Структуры	172
5.5. Дата и время.....	177
5.6. Класс строк.....	180
5.7. Перегрузка операторов.....	183
5.7.1. Математические операторы	184
5.7.2. Операторы сравнения	186
5.7.3. Операторы преобразования.....	187
5.8. Шаблоны	189
5.9. Анонимные типы	192
5.10. Кортежи.....	194
5.11. Форматирование строк.....	195
5.12. Динамика.....	196
Глава 6. Интерфейсы.....	198
6.1. Объявление интерфейсов	199
6.2. Реализация интерфейсов	201
6.3. Использование реализации интерфейса	202
6.4. Интерфейсы в качестве параметров.....	205
6.5. Перегрузка интерфейсных методов	206
6.6. Наследование	208
6.7. Клонирование объектов	209
6.8. Реализация по умолчанию	210
Глава 7. Массивы, списки, коллекции и словари	213
7.1. Базовый класс для массивов	213
7.2. Невыровненные массивы	215
7.3. Динамические массивы	217
7.4. Индексаторы массива.....	220
7.5. Интерфейсы массивов	221
7.5.1. Интерфейс <i>IEnumerable</i>	222
7.5.2. Интерфейсы <i>IComparer</i> и <i>IComparable</i>	225
7.6. Оператор <i>yield</i>	228
7.7. Стандартные списки	229
7.7.1. Класс <i>Queue</i>	229
7.7.2. Класс <i>Stack</i>	230
7.7.3. Класс <i>Hashtable</i>	231

7.8. Типизированные массивы	233
7.9. Словарь	236
Глава 8. Обработка исключительных ситуаций.....	238
8.1. Исключительные ситуации	239
8.2. Исключения в C#	241
8.3. Оформление блоков <i>try</i>	244
8.4. Ошибки в визуальных приложениях	245
8.5. Генерирование исключительных ситуаций	247
8.6. Иерархия классов исключений	248
8.7. Собственный класс исключения	249
8.8. Блок <i>finally</i>	252
8.9. Переполнение	253
8.10. Замечание о производительности	256
Глава 9. События	257
9.1. Делегаты	257
9.2. События и их вызов	258
9.3. Использование собственных делегатов	261
9.4. Делегаты изнутри	265
9.5. Анонимные методы	267
Глава 10. LINQ.....	268
10.1. LINQ при работе с массивами	268
10.1.1. SQL-стиль использования LINQ	268
10.1.2. Использование LINQ через методы	271
10.2. Магия <i>IEnumerable</i>	271
10.3. Доступ к данным	275
10.4. Использование LINQ для доступа к XML	276
Глава 11. Хранение информации	279
11.1. Файловая система	279
11.2. Текстовые файлы	282
11.3. Бинарные файлы	285
11.4. XML-файлы	289
11.4.1. Создание XML-документов	289
11.4.2. Чтение XML-документов	293
11.5. Потoki <i>Stream</i>	296
11.6. Потoki <i>MemoryStream</i>	297
11.7. Сериализация	299
11.7.1. Отключение сериализации	302
11.7.2. Сериализация в XML	304
11.7.3. Особенности сериализации	305
11.7.4. Управление сериализацией	307
Глава 12. Многопоточность	310
12.1. Класс <i>Thread</i>	311
12.2. Передача параметра в поток	314
12.3. Конкурентный доступ	316

12.4. Пул потоков.....	318
12.5. Домены приложений .NET	320
12.6. Ключевые слова <i>async</i> и <i>await</i>	323
12.7. Задачи или потоки — что выбрать?	330

Глава 13. Добро пожаловать в веб-программирование..... 332

13.1. Создание первого веб-приложения	332
13.2. Работа с конфигурацией сайта.....	339
13.3. Работа со статичными файлами	342
13.4. Модель – Представление – Контроллер	343
13.5. Маршрутизация.....	345
13.6. Подробнее про контроллеры	352
13.7. Представления	355
13.8. Модель представления	359
13.9. Представления Razor	362
13.9.1. Комментарии	362
13.9.2. Вывод данных в представлениях	363
13.9.3. C#-код.....	364
13.9.4. Условные операторы.....	365
13.9.5. Циклы	367
13.9.6. Исключительные ситуации.....	370
13.9.7. Ключевое слово <i>using</i>	370
13.10. Создание представлений	372
13.10.1. Макеты	372
13.10.2. Стартовый файл.....	376
13.10.3. Встраиваемые представления	378
13.10.4. Компоненты	382
13.10.5. Секции	385
13.11. Работа с формами	386
13.12. Проверка данных	391
13.13. Работа с сессиями	395
13.14. Cookie	397
13.15. Доступ к запросу.....	399

Глава 14. Базы данных..... 401

14.1. Библиотека ADO.NET	401
14.2. Строка подключения	403
14.3. Подключение к базе данных	405
14.4. Инъекция зависимостей на примере подключений	408
14.5. Пул соединений	412
14.6. Чтение данных из БД.....	413
14.7. Запросы с параметрами.....	417
14.8. Редактирование данных	420
14.9. Транзакции	421
14.10. Библиотека Dapper.....	422

Глава 15. Entity Framework Core 425

15.1. Создаем EF-приложение	426
15.2. Код первичен.....	427

15.3. Контекст базы данных.....	429
15.4. Доступ к базе данных	431
15.5. Миграции.....	433
15.6. База данных первична	439
15.7. LINQ	441
15.8. Связи: один ко многим	442
15.9. Работа со связями	444
15.10. Связи: многие ко многим	446
15.11. Связи: один к одному	451
15.12. Журналы	453
15.13. Как сохраняются данные?	456
15.14. Погружение в запросы	459
15.15. Асинхронный доступ.....	468
15.16. Поиск по шаблону	469
15.17. Удаление данных	470
15.18. Отслеживание изменений	471
15.19. Погружение в связи	475
Глава 16. Повторное использование кода	481
16.1. Библиотеки.....	481
16.2. Создание библиотеки	482
16.3. Приватные сборки	488
Глава 17. Сетевые сервисы	490
17.1. История.....	490
17.2. Создаем WebAPI.....	492
17.3. Теория Web API.....	496
17.4. Сессии.....	498
17.5. Полный цикл	500
Заключение.....	503
Список литературы.....	505
Приложение. Описание электронного архива, сопровождающего книгу.....	507
Предметный указатель	509

Предисловие

Хотя эта книга и носит название «Библия C#», посвящена она в целом платформе .NET от компании Microsoft. Частью этой платформы стали ОС, инструменты разработчика и .NET Framework — программный фреймворк для разработчика, чтобы мы могли писать приложения для платформы .NET. Надеюсь, я вас не запутал с первого же абзаца?

Фреймворк, если максимально упростить пояснение, — это библиотеки, предоставляющие языку программирования функциональность, которую мы можем использовать для написания программ. А на языке C# мы можем писать код, который будет использовать эту функциональность для создания программы, веб-сайта или любого другого приложения.

Первый .NET Framework появился в феврале 2002 года и на момент подготовки этого издания остается совсем чуть-чуть до его 20-летия, которое можно было бы отпраздновать, если бы не одно обстоятельство — в 2020 году на нем «поставили крест». Да, 2020 год оставил серьезный след в нашей истории. Вот и .NET Framework, просуществовав 19 лет, в том году «умер», но чтобы возродиться вновь...

Большинство языков программирования с богатой историей обладают одним большим недостатком. За время существования в них накапливается много устаревшего и небезопасного, но все это остается в языке для сохранения совместимости с уже написанным кодом. Разработка абсолютно нового языка позволила компании Microsoft избавиться от всего старого и создать нечто новое, чистое и светлое. Наверное, это слишком громкие слова, но сказать их тем не менее хочется.

В 2002 году язык программирования C# создавался практически с нуля специально, чтобы упростить с помощью .NET Framework разработку программ под платформу .NET, и язык этот сделали достаточно чистым и современным на тот период времени. Да и сейчас C# остается все еще современным и весьма популярным языком. Однако основной целью платформы .NET по факту была разработка приложений под ОС Windows. А во времена Билла Гейтса и Стива Балмера компания Microsoft не любила конкуренцию и не хотела поддерживать Linux или macOS, но времена меняются, и новое руководство Microsoft решило дружить с конкурентами.

Первое, что нужно было сделать ради зарождающейся дружбы, — расширить .NET с поддержки Windows еще и на Linux, и на macOS. Но вот только .NET Framework был заточен конкретно под Windows, и компанией было принято решение переписать его заново. Следуя этому решению, в 2016 году Microsoft начинает грандиозный проект, целью которого стало переписать фреймворк таким образом, чтобы приложения запускались не только на Windows, но и в других ОС, и этот проект вышел под именем .NET Core. В нем сохранялось все самое лучшее от .NET Framework — лучшие классы, лучшие методы — при максимальном желании обеспечить совместимость и сделать все так, чтобы и сам .NET стал лучше.

Всего вышло три версии .NET Core, в результате чего он практически догнал своего предшественника .NET Framework по части возможностей написания веб-приложений. И тогда Microsoft принимает решение — .NET Framework далее не развивать. Его будут поддерживать, исправлять возможные баги, но новых версий ожидать не стоит, последней версией так и останется 4.8. Основным же фреймворком будет теперь .NET Core, ставший настолько продвинутым, что ему поменяли даже имя, отбросив «Core», — теперь это просто .NET.

Итак, .NET Framework просуществовал до версии 4.8, .NET Core — до версии 3.1. На смену им пришел фреймворк .NET 5. Фактически он представляет собой продолжение ветки .NET Core, но с учетом того, что «Core» более в его названии не фигурирует, мы сразу же можем сказать, что это — наше будущее.

Новый фреймворк .NET действительно очищен от многих ошибок и проблем прошлого, потому что основной язык C# создан относительно недавно, да и библиотеки кода были не так давно полностью переписаны в соответствии с современными реалиями.

Фреймворк .NET сделан максимально совместимым с .NET Framework, чтобы программисты могли легко на него мигрировать и не потерять наработки, которые они создали за последние 19 лет. Если вы знаете .NET Framework, то вам не придется переучиваться. Эта книга изначально была написана во время существования .NET Framework, и мне практически не пришлось изменять включенный в нее код, несмотря на выход .NET Core и .NET 7, потому что большинство изменений произошли внутри фреймворка за счет его переписывания и оптимизации. Из него убрали все зависимости с ОС Windows и сделали его универсальным для возможности работы с ним в Linux и macOS.

Чтобы понять, почему мне нравятся .NET и C#, давайте выделим их реальные преимущества. Итак, к основным их преимуществам я бы отнес:

- ❑ *универсальный API* — на каком бы языке вы ни программировали, вам предоставляются одни и те же имена классов и методов. Еще недавно это являлось большим преимуществом, потому что для .NET можно было писать не только на C#, но и на Visual Basic или C++, но Visual Basic «заморозили» и, по последним данным, развивать его больше не станут. Его не «убивают», но новых возможностей в него, скорее всего, добавлять не будут;
- ❑ *защищенный код* — платформу Win32 очень часто ругали за ее незащищенность. В ней, действительно, есть очень слабое звено — незащищенность кода и воз-

можность перезаписывать любые участки памяти. Самым страшным в Win32 была работа с массивами, памятью и со строками (последние являются разновидностью массива). С одной стороны, это предоставляет мощные возможности системному программисту, который умеет правильно распоряжаться памятью. С другой стороны, в руках неопытного программиста такая возможность превращается в уязвимость. Сколько раз нам приходилось слышать об ошибках переполнения буфера из-за неправильного выделения памяти? Уже и сосчитать сложно.

На платформе .NET вероятность такой ошибки стремится к нулю — если вы используете управляемый код, и Microsoft не допустит ошибок при реализации самой платформы.

Впрочем, платформа .NET не является абсолютно безопасной, потому что существуют не только ошибки переполнения буфера — есть еще и ошибки логики работы программы. А такие ошибки являются самыми опасными, и от них нет «волшебной таблетки»;

- *полная ориентированность на объекты.* Объектно-ориентированное программирование (ООП) — это не просто дань моде, это мощь, удобство и скорость разработки. Платформа Win32 все еще основана на процедурах и наследует все недостатки этого подхода. Любые объектные надстройки — такие как Object Windows Library (OWL), Microsoft Foundation Classes (MFC) и др. — решают далеко не все задачи;
- *сборка мусора* — начинающие программисты очень часто путаются с тем, когда нужно выделить память, когда освободить память, как правильно освободить всю выделенную память, а когда это делать не обязательно или даже вредно. Приходится долго объяснять, что такое локальные и глобальные переменные, распределение памяти, стек и т. д. А ведь если память освободить раньше, чем надо, то обращение к несуществующим объектам приведет к краху программы.

На платформе .NET за уничтожение объектов, хотя вы и можете косвенно повлиять на этот процесс, отвечает сама платформа. Утечки памяти все еще возможны, но чтобы это произошло, нужно уже немного постараться;

- *распределенные вычисления* — платформа .NET ускоряет разработку приложений с распределенными вычислениями, что достаточно важно для корпоративного программного обеспечения. В качестве транспорта при взаимодействии используются технологии HTTP¹, XML², SOAP³, REST⁴ и великолепная и быстрая WCF (Windows Communication Foundation), которая позволяет связывать сервисы различными протоколами;

¹ HTTP, HyperText Transfer Protocol — протокол передачи гипертекстовых файлов.

² XML, Extensible Markup Language — расширяемый язык разметки.

³ SOAP, Simple Object Access Protocol — простой протокол доступа к объектам.

⁴ REST, Representational state transfer — если дословно, то это репрезентативная передача состояния.

- *открытость стандартов* — при рассмотрении предыдущего преимущества мы затронули открытые стандарты HTTP, XML, SOAP и REST. Открытость — это неоспоримое преимущество, потому что предоставляет разработчику большую свободу. Платформа .NET является открытой, и ее может использовать кто угодно.

Список можно продолжать и дальше, но уже видно, что будущее есть. Каким станет это будущее — пока еще большой и сложный вопрос. Но, глядя на средства, которые были вложены в разработку и рекламную кампанию платформы .NET, можно предположить, что Microsoft не упустит своего и сделает все возможное для обеспечения ее долгой и счастливой жизни.

Я знаком с платформой .NET с момента появления ее первой версии и пытался изучать ее, несмотря на то, что синтаксис и возможности платформы, как уже было отмечено ранее, не вызывали у меня восторга. Да, язык C# вобрал в себя все лучшее от C++ и Delphi, но все равно работа с ним особого удовлетворения поначалу не приносила. Тем не менее я продолжал его изучать в свободное время, т. к. люблю находиться на гребне волны, а не стоять в очереди за догоняющими.

Возможно, мое первоначальное отрицательное отношение к .NET было связано со скудными возможностями самой среды разработки Visual Studio, потому что после выхода финальной версии Visual Studio .NET и версии .NET Framework 2.0 мое отношение к языку начало меняться к лучшему. Язык и среда разработки стали намного удобнее, а код — читабельным, что очень важно при разработке больших проектов.

А с появлением .NET 5 и полноценной поддержки macOS и Linux я теперь могу писать приложения на моем любимом C# на MacBook с помощью среды разработки Visual Studio для macOS или более простого и быстрого редактора кода Visual Studio Code.

В настоящее время мое личное отношение к языку и среде разработки Visual Studio .NET более чем положительное. Для меня эта платформа стала номером один в моих личных проектах.

Благодарности

Я хочу в очередной раз поблагодарить свою семью, которая была вынуждена терпеть мое отсутствие, пока я сидел над книгой.

Я благодарю всех тех, кто помогал мне в работе с ней в издательстве «БХВ»: редакторов и корректоров, дизайнеров и верстальщиков — всех, кто старался сделать эту книгу интереснее для читателя.

Спасибо всем моим друзьям — тем, кто меня поддерживал, и всем тем, кто уговорил меня все же написать эту книгу, а это в большинстве своем посетители моего сайта **www.flenov.info**.

Бонус

Я долго не хотел браться за этот проект, имея в виду, что люди в наше время не покупают книги, а скачивают их из Интернета, да и тратить свободное время, отрывая его от семьи, тяжело. Очень жаль, что люди не покупают книг. Издательство «БХВ» устанавливает на свои издания не такие уж высокие цены, и можно было бы отблагодарить всех тех людей, которые старались для читателя, небольшой суммой. Если вы скачали книгу, ознакомились с ней, и она вам понравилась, то я советую вам купить ее полноценную «бумажную» версию.

Я же со своей стороны постарался сделать книгу максимально интересной, а на сервере издательства «БХВ» мы выложили сопровождающую ее дополнительную информацию в виде статей и исходных кодов для дополнительного улучшения и совершенствования ваших навыков (см. *приложение*).

Электронный архив с этой информацией можно скачать по ссылке <https://zip.bhv.ru/9785977518284.zip> или со страницы книги на сайте издательства <https://bhv.ru/>.

Структура книги

В своих предыдущих книгах я старался как можно быстрее приступить к показу визуального программирования в ущерб начальным теоретическим знаниям. В этой же я решил потратить немного времени на погружение в теоретические глубины искусства программирования, чтобы потом рассмотрение визуального программирования пошло у вас как по маслу. А чтобы теория вводной части не казалась вам слишком скучной, я стараюсь здесь подносить ее как можно интереснее и по возможности придумывать занимательные примеры.

В первых четырех главах мы станем писать программы, не имеющие графического интерфейса, — информация будет выводиться в консоль. В мире, где властвует графический интерфейс: окна, меню, кнопки и панели — консольная программа может выглядеть несколько дико. Но командная строка еще жива — она, наоборот, набирает популярность, и ярким примером тому является PowerShell (это новая командная строка, которая вошла в состав Windows, начиная с версии Server 2008, и может быть установлена в Windows 8/10/11).

К тому же есть еще и веб-программирование, где нет собственного визуального интерфейса, а только специальная разметка, которая выполняется в браузере. Я последние 16 лет работаю в веб-индустрии и создаю сайты на языке C#. Мне так же очень часто приходится писать разные консольные утилиты, предназначенные для импорта/экспорта данных из баз, их конвертации и обработки, создания отчетов и т. п.

В предыдущих изданиях книги я много внимания уделял разработке пользовательских приложений, или, как по-другому их часто называют, *desktopных*. В пятом издании я решил все переписать в очередной раз с упором на веб-программирование. Если вас интересуют пользовательские приложения, о которых шла речь в предыдущих изданиях, то эта информация не исчезла — она все еще существует на моем сайте по адресу: <https://www.flenov.info/books/read/biblia-csharp>. В шестом же издании я решил добавить больше деталей.

Я не пытался переводить файл справки, потому что Microsoft уже сделала это для нас. Моя задача — научить вас понимать программирование и уметь строить код. То есть попробовать рассказать про C# максимально простым языком. А за под-

робным описанием классов, методов и т. п. всегда можно обратиться к MSDN (Microsoft Development Network) — обширной библиотеке технической информации, расположенной на сайте **www.msdn.com** и доступной на русском языке, — там имеется перевод всех классов, методов и свойств с подробным описанием и простыми примерами.

В третьем издании книги я исправил некоторые имевшиеся ошибки предыдущих изданий и добавил описание новых возможностей, которые Microsoft представила миру в .NET 3.5, 4.0 и 4.5, а также в .NET Core.

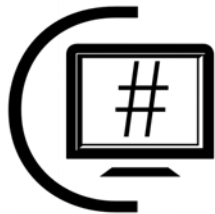
В четвертом издании я убрал все, что касается приложений с пользовательским интерфейсом, и переписал все примеры с использованием последней версии .NET Core и возможностей веб-программирования. Я обратил там больше внимания на .NET Core, потому что этот стандарт позволял писать .NET-код, который может выполняться на разных платформах.

В пятом издании все внимание уделяется фреймворку .NET 5, который является наследником .NET Core. Это как бы следующая реинкарнация платформы .NET Framework, в которой Microsoft сделала свой популярный фреймворк еще лучше. Это не значит, что старый код можно забыть и .NET Framework «мертв». Весь существующий код никуда не делся, его просто стандартизируют. Но более детально с .NET мы начнем знакомиться, начиная уже со следующей главы.

В этом, шестом издании книга сильно похудела, потому что я убрал все про десктопное программирование, и у меня высвободилось значительное пространство для того, чтобы добавить больше деталей, больше информации и рассказать про обновленный фреймворк .NET 7.

Итак, пора переходить к более интересным вещам и начать погружение в мир .NET.

ГЛАВА 1



Введение в .NET

Платформа Microsoft .NET состоит из набора базовых классов и *общезыковой среды выполнения* (Common Language Runtime, CLR). Базовые классы, которые входят в состав .NET, поражают своей мощностью, универсальностью, удобством использования и разнообразием.

Я постараюсь дать только минимум необходимой вводной информации, чтобы как можно быстрее перейти к изучению языка C# и к самому программированию. Я люблю делать упор на практику и считаю ее наиболее важной в нашей жизни. А по ходу практических занятий мы будем ближе знакомиться с теорией.

Для чтения этой и последующих глав рекомендуется иметь установленную среду разработки Visual Studio, которую можно скачать с сайта по адресу: **<https://visualstudio.microsoft.com>**.

Для всех примеров достаточно даже бесплатной версии среды разработки Visual Studio Community Edition, которую можно свободно скачать по адресу: **<https://visualstudio.microsoft.com/ru/vs/community/>**. И хотя эта версия действительно бесплатная, с ее помощью можно делать даже большие проекты. Если вы программист-одиночка, то можете использовать версию Community Edition для создания как бесплатных, так и платных приложений. Для корпоративных разработчиков имеются ограничения, с которыми лучше познакомиться на сайте среды.

1.1. Платформа .NET

В *предисловии* мы уже затронули основные преимущества .NET, а сейчас рассмотрим эту платформу более подробно. До недавнего времени ее поддерживала только Microsoft, но на C# можно было писать программы для Linux (благодаря Mono¹) и

¹ Mono — проект по созданию полноценного воплощения системы .NET Framework на базе свободного программного обеспечения.

даже игры на Unity¹. Однако все эти реализации немного отличались друг от друга — если написать программу для Windows, то это не значит, что она заработает в Mono на Linux или где-либо еще.

Я не могу знать реальных причин принятия решений внутри компании Microsoft, но мне кажется, что на появление .NET Core повлияло желание перенести .NET на другие платформы. Core по-английски — это *ядро*. Изначально в .NET включили все, что необходимо для написания полноценных приложений, но в каждой ОС своя файловая система, да и многие моменты реализованы по-разному. Нельзя было просто взять и перенести на все платформы имеющиеся функции .NET, и вместо этого было создано ядро, которое точно будет работать на всех платформах.

Когда фреймворк .NET Core вырос и по некоторым возможностям начал обходить существующий .NET Framework, в компании Microsoft решили избавиться от такой путаницы и как бы объединить все усилия в одну ветку. Так что вместо двух веток мы получили одну, и ей дали название просто .NET, хотя на самом деле она является продолжением .NET Core.

При написании консольных приложений на .NET или на старом добром .NET Framework вы особой разницы не заметите. А вот для веб-программирования разница есть, и если начинать новый веб-проект, то я бы делал его на .NET, потому что архитектуру написания веб-приложений изменили весьма сильно. Новая архитектура намного лучше, быстрее и в ней используются современные подходы (паттерны) программирования.

Не знаю, насколько для вас это является преимуществом, но исходные коды .NET открыты (т. е. это проект OpenSource), и их можно найти на сайте GitHub по адресу: <https://github.com/dotnet>. Мне лично этот факт не важен, потому что в исходные коды .NET я заглядывать не планирую. Но если вам интересно на них посмотреть, вы можете это сделать, а если у вас еще и достаточно опыта, то можно попробовать даже что-то в них улучшить и предложить свое улучшение.

В этой книге мы будем изучать язык программирования C#, который является основным языком для платформы .NET.

1.1.1. «Кубики» .NET

Если отбросить всю рекламу, которую нам предлагают, и взглянуть на проблему глазами разработчика, то .NET описывается следующим образом: в среде разработки Visual Studio вы можете с использованием платформы .NET разрабатывать приложения любой сложности, которые очень просто интегрируются с серверами и сервисами от Microsoft.

Основа всего, центральное звено платформы — это фреймворк .NET. Давайте посмотрим на основные составляющие этой платформы:

- *среда выполнения Common Language Runtime (CLR)*. CLR работает поверх ОС — это и есть виртуальная машина, которая обрабатывает IL-код² программы. Код

¹ Unity — межплатформенная среда разработки компьютерных игр.

² IL, Intermediate Language — промежуточный язык.

IL — это аналог бинарного кода для платформы Win32 или байт-кода для виртуальной машины Java. Во время запуска приложения IL-код на лету компилируется в машинный код под то «железо», на котором запущена программа. Да, сейчас практически все работает на процессорах Intel, но никто не запрещает реализовать платформу на процессорах других производителей;

- *базовые классы .NET* — используемые, пишете вы приложение, не зависящее от платформы, или для Windows. Как и библиотеки на других платформах, здесь нам предлагается обширный набор классов, которые упрощают создание приложения. С помощью таких компонентов вы можете строить свои приложения как бы из блоков;
- *расширенные классы .NET*. В предыдущем пункте говорилось о базовых классах, которые реализуют базовые возможности. Также выделяют и более сложные компоненты доступа к базам данных, XML и др.

Надо также понимать, что .NET не является переработкой Java, — у них общее только то, что обе платформы являются виртуальными машинами и выполняют не машинный код, а байт-код. Да, обе они произошли от C++, но каждая идет своим путем.

1.1.2. Сборки

Термин *сборки* в .NET переводят и преподносят по-разному. Я встречал много различных описаний и переводов — например: *компоновочные блоки* или *бинарные единицы*. На самом деле, если не углубляться в терминологию, а посмотреть на сборки глазами простого программиста, то окажется, что это просто файлы, являющиеся результатом компиляции. Именно *конечные файлы*, потому что среда разработки может сохранить на диске после компиляции множество промежуточных файлов.

Наиболее распространены два типа сборок: библиотеки, которые сохраняются в файлах с расширением `dll`, и исполняемые файлы, которые сохраняются в файлах с расширением `exe`. Несмотря на то что расширения файлов такие же, как и у Win32-библиотек и приложений, это все же совершенно разные файлы по своему внутреннему содержанию. Программы .NET содержат не инструкции процессора, как классические Win32-приложения, а IL-код¹. Этот код создается компилятором и сохраняется в файле. Когда пользователь запускает программу, то она на лету компилируется в машинный код и выполняется на процессоре.

Так что я не буду здесь использовать мудреные названия типа «компоновочный блок» или «бинарная единица», а просто стану называть вещи простыми именами: исполняемый файл, библиотека и т. д. — в зависимости от того, что мы компилируем. А если нужно будет определить эти вещи общим названием, не привязываясь к типу, я просто буду говорить: «сборка».

¹ Как уже отмечалось ранее, IL-код — это промежуточный язык (Intermediate Language). Можно также встретить термины CIL (Common Intermediate Language) или MSIL (Microsoft Intermediate Language).

Благодаря тому, что ПЛ-код не является машинным, а интерпретируется JIT-компилятором¹, говорят, что *код управляет* этим JIT-компилятором. Машинный код выполняется напрямую процессором, и ОС не может управлять этим кодом. А вот ПЛ-код выполняется на платформе .NET, и уже она решает, как его выполнять, какие процессорные инструкции использовать, а также берет на себя множество рутинных вопросов безопасности и надежности выполнения.

Тут нужно пояснить, почему программы .NET внешне не отличаются от классических приложений и файлы их имеют те же расширения. Так сделали, чтобы скрыть сложности реализации от конечного пользователя. Зачем ему знать, как выполняется программа и на чем она написана? Это для него совершенно не имеет значения. Как я люблю говорить, главное — это качество программы, а как она реализована, на каком языке и на какой платформе, нужно знать только программисту и тем, кто этим специально интересуется. Конечного пользователя это интересовать не должно и на самом деле не интересует.

Помимо кода, в сборке хранится информация (метаданные) о задействованных типах данных. Метаданные сборки очень важны с точки зрения описания объектов и их использования. Кроме того, в файле хранятся метаданные не только данных, но и самого исполняемого файла, описывающие версию сборки и содержащие ссылки на подключаемые внешние сборки. В последнем утверждении кроется одна интересная мысль — сборки могут ссылаться на другие сборки, что позволяет нам создавать многомодульные сборки (проще говоря, программы, состоящие из нескольких файлов).

Что-то я вдруг стал использовать много заумных слов, хотя постоянно стараюсь давать простые пояснения... Вот, например, *метаданные* — это просто описание чего-либо. Таким описанием может быть структурированный текст, в котором указано, что находится в исполняемом файле, какой он версии, и т. п.

Чаще всего код делят на сборки по принципу логической завершенности. Допустим, наша программа реализует работу автомобиля. Все, что касается работы двигателя, можно поместить в отдельный модуль, а все, что касается трансмиссии, — в другой. Помимо этого, каждый модуль будет разбивать составляющие двигателя и трансмиссии на более мелкие составляющие с помощью классов.

Теперь еще на секунду хочу вернуться к JIT-компиляции и сказать об одном сильном преимуществе этого метода. Когда пользователь запускает программу, то она компилируется так, чтобы максимально эффективно выполняться на «железе» и ОС компьютера, где сборка была запущена на выполнение. Для персональных компьютеров существует множество различных процессоров, и, компилируя программу в машинный код, чтобы он выполнялся на всех компьютерах, программисты очень часто — чтобы не сужать рынок продаж — не учитывают современные инструкции процессоров. А вот JIT-компилятор может учесть эти инструкции и оптимально скомпилировать программу под конкретный процессор, установленный на кон-

¹ Just-In-time Compiler — компилятор периода выполнения.

кретном устройстве. Кроме того, разные ОС обладают разными функциями, и JIT-компилятор также может использовать возможности ОС максимально эффективно.

Делает ли такую оптимизацию среда выполнения .NET — я не знаю. Из информации, доступной в Интернете, все ведет к тому, что некоторая оптимизация имеет место. Но достоверно мне это не известно. Впрочем, утверждают, что магазин приложений Windows может перекомпилировать приложения перед тем, как отправлять их пользователю. И когда вы через этот магазин скачиваете что-либо, то получаете специально скомпилированную под ваше устройство версию.

Из-за компиляции кода во время запуска первый его запуск на компьютере может оказаться весьма долгим. Но последующее выполнение будет намного быстрее.

1.2. Обзор среды разработки Visual Studio .NET

Познакомившись с основами платформы .NET, перейдем непосредственно к практике и бросим беглый взгляд на новую среду разработки — посмотрим, что она предоставляет нам, как программистам.

Мы познакомимся здесь с Visual Studio 2022 Community Edition, но работа с ней не отличается от работы с любой другой версией Visual Studio.

Далее в этой главе я попытался подробно предоставить базовую информацию о работе с Visual Studio, но некоторые вещи лучше один раз увидеть, чем подробно прочитать, поэтому на моем сайте я опубликовал видео, в котором показываю базовые основы работы с этой средой разработки, — это видео можно найти здесь: <https://www.flenov.info/books/show/biblia-csharp>. Видео и текстовая информация дополняют друг друга, поэтому я рекомендую вам и прочитать книгу, и просмотреть видео, а не ограничиваться чем-то одним.

1.2.1. Работа с проектами и решениями

В Visual Studio любая программа заключается в *проект*. Проект — это как каталог для файлов. Он обладает определенными свойствами (например, платформой и языком, для которых создан проект) и может содержать файлы с исходным кодом программы, который необходимо скомпилировать в исполняемый файл. Проекты могут объединяться в *решения* (Solution). Более подробную информацию о решениях и проектах можно найти в файле Documents\Visual Studio 2008.docx из сопровождающего книгу электронного архива (см. *приложение*).

После запуска Visual Studio может появиться окно приветствия, в котором слева будут расположены проекты, с которыми вы работали ранее, а справа — кнопки для выполнения базовых команд (рис. 1.1). Самая нижняя кнопка **Create a New Project** позволяет создать новый проект, а имеющаяся чуть ниже ее ссылка **Continue without code** (Продолжить без кода) открывает среду разработки без создания или открытия проекта.

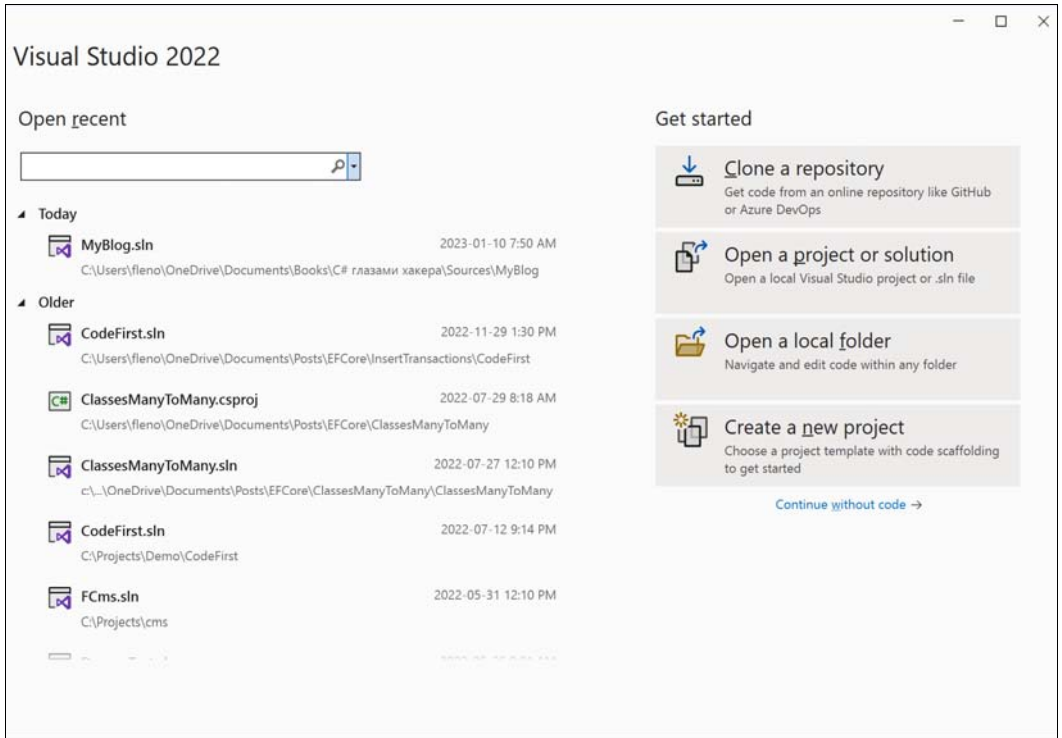


Рис. 1.1. Окно приветствия Visual Studio

Для создания нового проекта выберите в меню Visual Studio команду **File | New | Project** (Файл | Новый | Проект). Перед вами откроется окно **New Project** (Новый проект), в левой части которого расположены шаблоны проектов, которые вы недавно использовали (рис. 1.2). Справа сверху расположены три выпадающих списка:

- ❑ среда разработки Visual Studio может работать и компилировать проекты на нескольких языках: Visual C++, Visual C#, Visual Basic и т. д., и в первом выпадающем списке можно выбрать язык программирования;
- ❑ из второго списка следует выбрать платформу, под которую вы хотите создать приложение. В Visual Studio можно писать десктопные приложения для Windows, Android, iOS, Linux, Azure и т. д.
- ❑ третий список предоставляет выбор типа приложения: мобильное, веб-приложение, игры, консольное приложение и т. д.

При выборе значений из этих выпадающих списков набор доступных шаблонов справа внизу окна будет изменяться. Мы рассматриваем C#, поэтому для всех проектов этой книги в первом списке выбираем **C#**. Во втором можно оставить **All Platforms**, а в третьем выберем **Console** для первых приложений, а когда доберемся до веб-приложений, то можно будет выбрать **Web**, чтобы быстрее найти нужный нам шаблон.

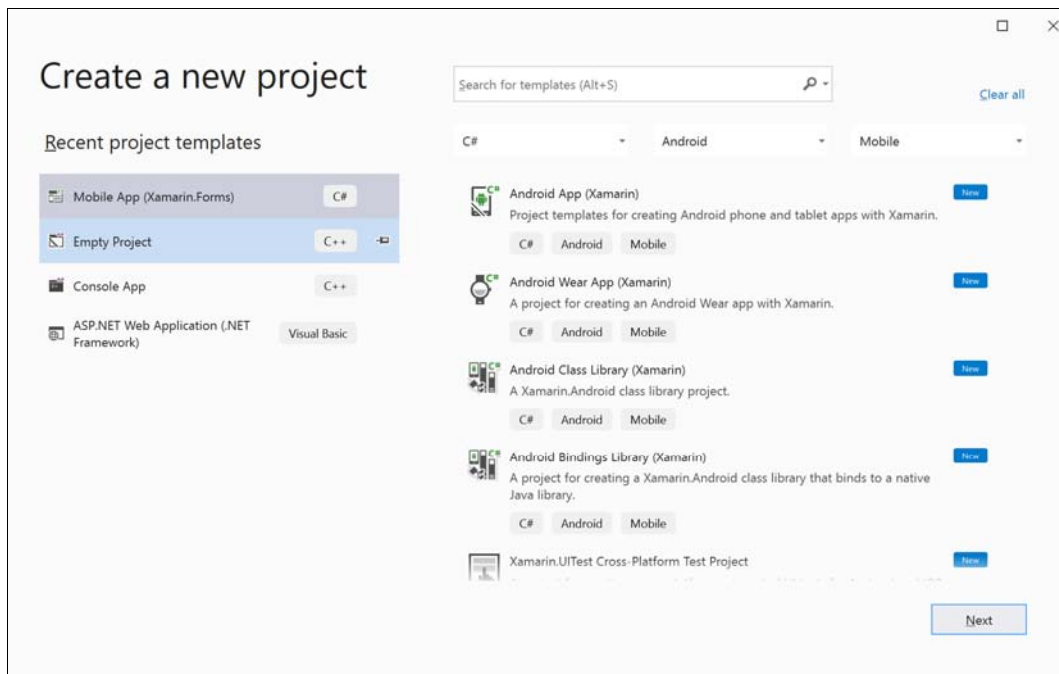


Рис. 1.2. Окно создания нового проекта

Разобравшись со списками, нажимаем кнопку **Next** (Далее), и на втором шаге создания проекта нам предложат ввести три параметра:

- ❑ **Project Name** — здесь вы указываете имя будущего проекта;
- ❑ **Location** — расположение каталога проекта.
- ❑ **Solution name** — имя решения (по умолчанию соответствует имени проекта и, возможно, это поле будет неизменяемым).

Давайте сейчас остановимся на секунду на третьем пункте — имени решения. Если вы работаете над большим проектом, то он может состоять из нескольких файлов: основного исполняемого файла и динамических библиотек DLL, в которых хранится код, которым приложения могут делиться даже с другими приложениями.

Объединяя библиотеки и исполняемые файлы в одно большое решение, мы упрощаем компиляцию — создание финальных файлов. Мы просто компилируем все решение, а среда разработки проверит, какие изменения произошли, и в зависимости от этого перекомпилирует те проекты (библиотеки или исполняемые файлы), которые изменились.

Если мы, например, хотим создать приложение Машина, то начнем с создания приложения и решения Машина, а потом можем к этому решению добавить дополнительные проекты: двигатель, коробка передач и т. д. Именно поэтому имя решения по умолчанию может быть неизменяемым и соответствовать имени проекта.

В окне второго шага имеется также флажок **Please solution and project in the same directory** (Поместить решение и проект в один каталог), и если вы планируете по-

мещать все в одну папку, то это скажет мастеру создания проекта, что вы не планируете добавлять новые проекты, и именно поэтому поле ввода имени решения будет недоступно. Но если вы сбросите этот флажок, то для проекта будет создана отдельная папка. Поскольку вы решили завести отдельную папку для проекта, то, возможно, вы уже планируете создавать дополнительные проекты, а значит, имя решения может быть иным, и вам позволят его сменить.

При задании имени и расположения проекта будьте внимательны. Допустим, что в качестве пути (в поле **Location**) вы выбрали `C:\Projects`, а имя (в поле **Name**) назначили `MyProject`. Созданный проект тогда будет находиться в каталоге `C:\Projects\MyProject`. То есть среда разработки создаст каталог с именем проекта, указанным в поле **Name**, в каталоге, указанном в поле **Location**.

Давайте создадим новый пустой C#-проект, чтобы увидеть, из чего он состоит. В окне создания нового проекта (рис. 1.3) выбираем из первого выпадающего списка **C#**, во втором оставляем все по умолчанию, а из третьего выбираем **Console**. В списке шаблонов у вас должно остаться только **Console Application** (это приложение .NET) и **Console App** (приложение .NET Framework). В скобках при **Console App** вы можете видеть **.NET Framework**, что как раз указывает на то, что это старый фреймворк. Нас же интересует первый из этих шаблонов. Под именем шаблона показано, что он подходит для консольного приложения, которое потом сможет запускаться на Linux, macOS и Windows.

Нажимаем в этом окне кнопку **Next** и на следующем шаге вводим имя проекта — `TestApplication`, а место расположения можно оставить по умолчанию. На сле-

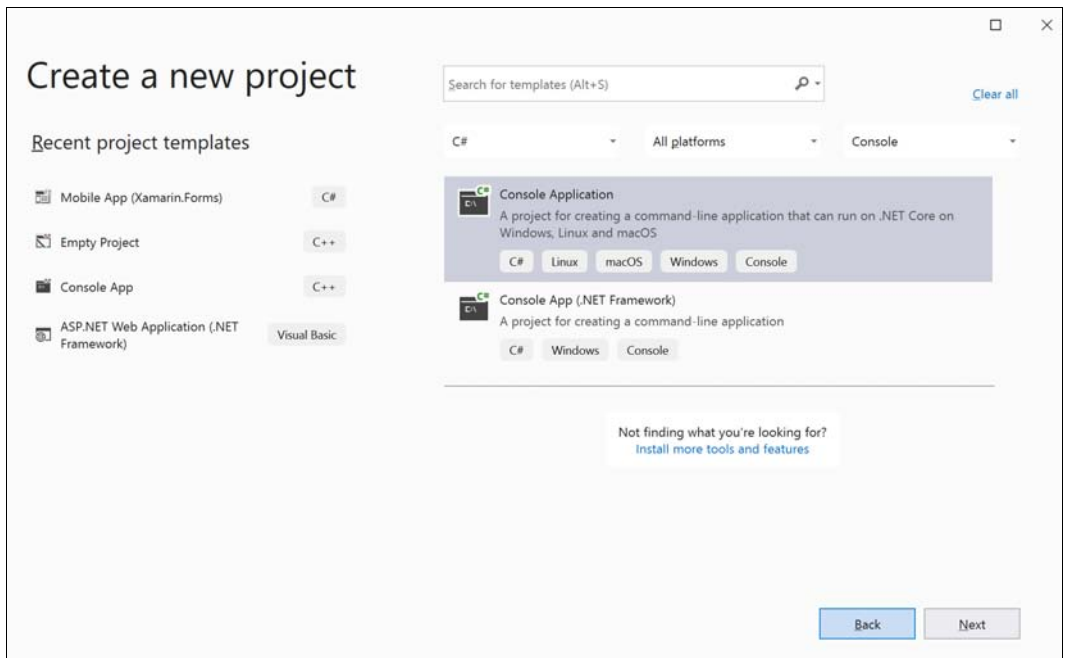


Рис. 1.3. Создаем консольное приложение

дующем шаге (рис. 1.4) нам нужно выбрать целевой фреймворк (поле **Framework**) — давайте выберем наиболее свежий, а на момент подготовки этого издания им является **.NET 7**.

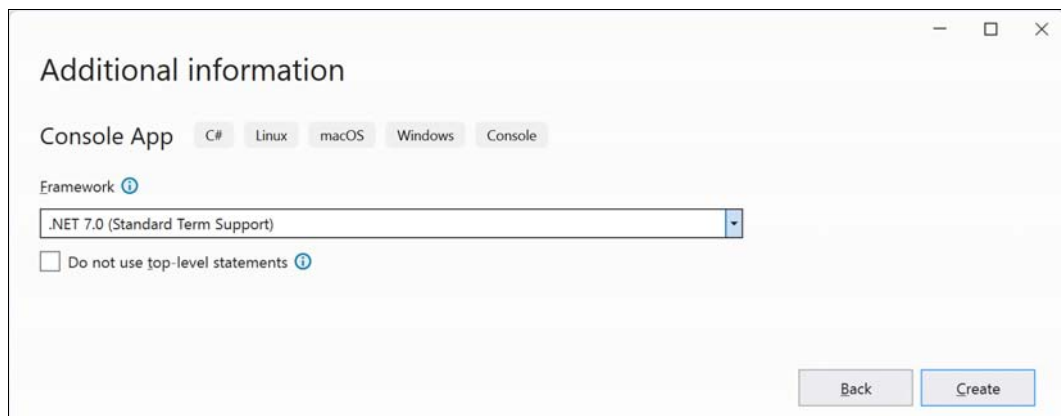


Рис. 1.4. Выбор версии .NET

В этом же окне в последней версии Visual Studio появился новый флажок: **Do not use top-level statements** (Не использовать код верхнего уровня). Установите его. Это новая функция, которая может ввести вас в замешательство, потому что в этом случае от нас будет скрыт код, который лучше все же на начальном этапе увидеть.

Указав имя проекта и его расположение, для завершения создания нового проекта нажмите кнопку **Create** (Создать). В результате вы увидите окно среды разработки, показанное для нашего проекта на рис. 1.5. Оно включает несколько окон, и одно из главных здесь — это **Solution Explorer** (Проводник решения), которое расположено вдоль правой кромки окна среды разработки. Если по какой-либо причине вы его не видите (оно может быть закрыто), то для его открытия надо выбрать пункт меню **View | Solution Explorer**.

В окне **Solution Explorer** самая первая запись — имя решения. Чуть ниже в дереве элементов вы увидите имя проекта, которое как бы вложено в решение. В папке с проектом показана папка **Dependencies** (Зависимости) и файлы, которые входят в проект. О зависимостях мы еще поговорим далее — сейчас же мы только знакомимся с проектом. Из файлов пока доступен лишь один — **Program.cs**. Это файл, где мы будем впоследствии писать код. Имена файлов с кодом имеют расширение **cs**, что означает «C Sharp» и соответствует языку программирования **C#**, который рассматривается в этой книге.

Если щелкнуть правой кнопкой мыши на имени проекта и в выпадающем меню выбрать **Open folder in file explorer**, то откроется окно проводника и папка, в которой находятся файлы проекта (рис. 1.6).

Здесь мы видим файл нашего решения, который имеет расширение **sln** (от solution, решение), файл проекта с расширением **csproj** (C Sharp Project, проект C#) и файл с исходным кодом **Program.cs**, который мы видели и в окне **Solution Explorer**.

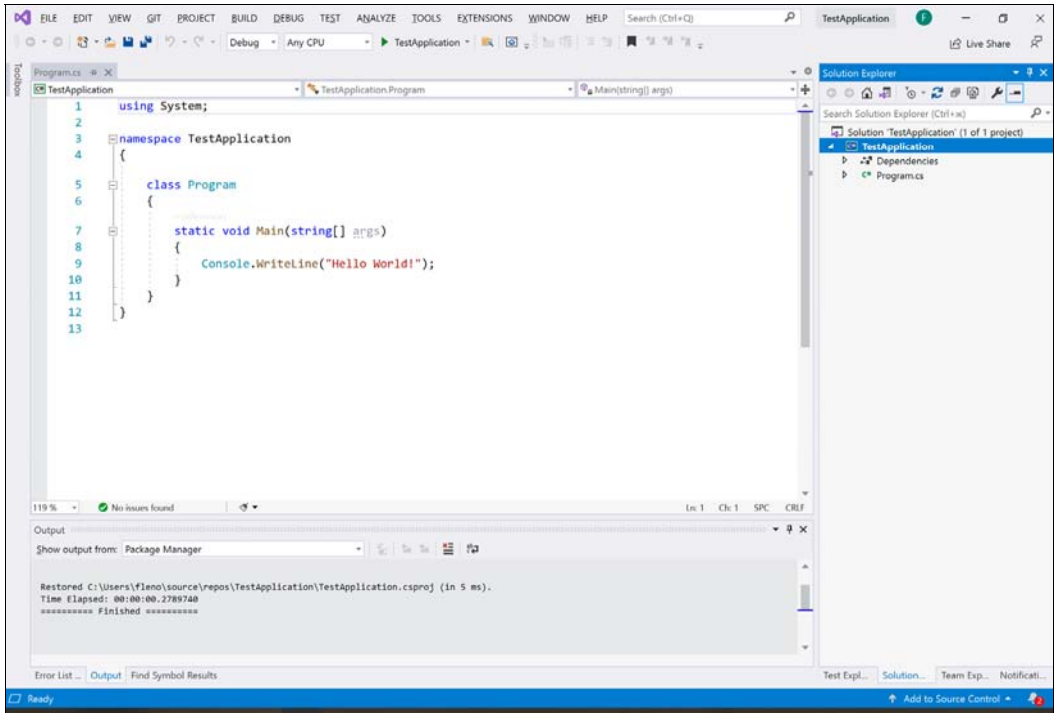


Рис. 1.5. Окно среды разработки с открытым проектом

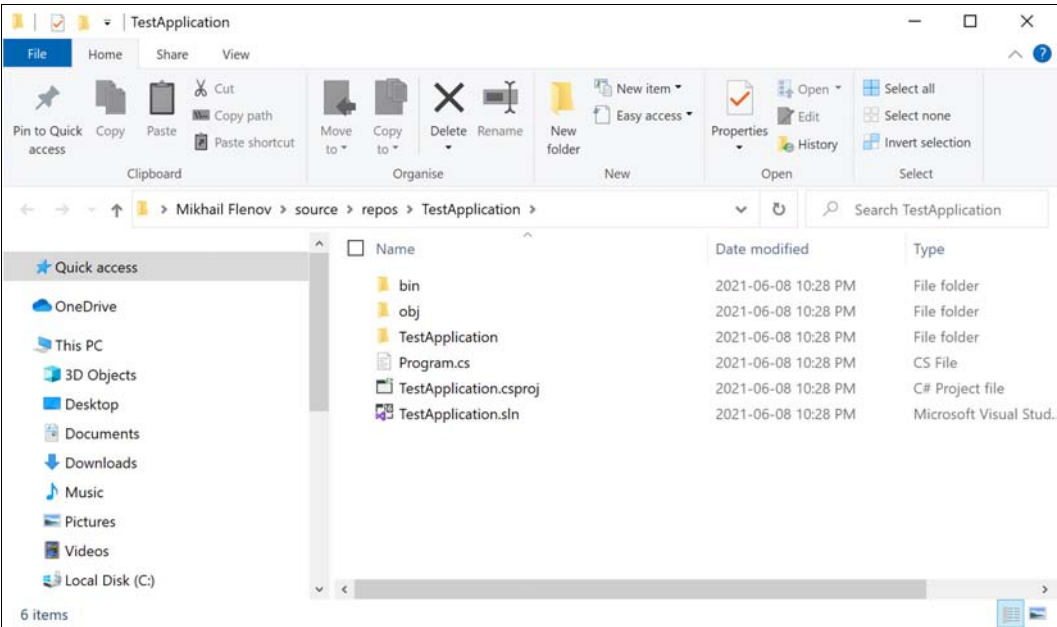


Рис. 1.6. Файлы проекта

Файл проекта, в котором находятся все настройки и описания входящих в проект файлов, как уже отмечено, имеет расширение `csproj`. Этот файл создается в формате XML, и его легко просмотреть в любом текстовом редакторе. В текстовом редакторе вы даже можете его редактировать, но в большинстве случаев проще использовать для этого специализированные диалоговые окна, которые предоставляет среда разработки.

Чтобы посмотреть или изменить свойства проекта, можно щелкнуть на имени проекта в окне **Solution Explorer** и выбрать из появившегося меню пункт **Properties** (Свойства).

Чтобы открыть файл для редактирования, достаточно щелкнуть на нем мышью двойным щелчком в окне **Solution Explorer**.

Для переименования решения щелкните на его имени в дереве правой кнопкой мыши и в контекстном меню выберите пункт **Rename** (Переименовать). Таким же образом можно переименовывать и файлы.

Для добавления проекта в решение щелкните на имени решения в дереве правой кнопкой мыши и в контекстном меню выберите **Add** (Добавить). В этом меню также можно увидеть следующие пункты:

- ❑ **New Project** — создать новый проект. Перед вами откроется окно создания нового проекта, который будет автоматически добавлен в существующее решение;
- ❑ **Existing Project** — добавить существующий проект. Этот пункт удобен, если у вас уже есть проект и вы хотите добавить его в это решение;
- ❑ **Existing Web Site** — добавить существующий проект с веб-сайта;
- ❑ **Add New Item** — добавить новый элемент в решение, а не в отдельный проект. Не так уж и много типов файлов, которые можно добавить прямо в решение. В диалоговом окне выбора файла вы увидите в основном различные типы текстовых файлов и картинки (значки и растровые изображения);
- ❑ **Add Existing Item** — добавить существующий элемент в решение, а не в отдельный проект.

Чтобы создать исполняемые файлы для всех проектов, входящих в решение, щелкните правой кнопкой мыши на имени решения и в контекстном меню выберите или **Build Solution** (Собрать решение) — компилироваться будут только измененные файлы, или **Rebuild Solution** (Полностью собрать проект) — компилироваться будут все файлы. Такие же пункты меню есть и в основном меню **Build**. Для сборки проекта проще использовать комбинацию клавиш `<Ctrl>+<Shift>+<B>`.

Если же щелкнуть правой кнопкой мыши на имени проекта, то в контекстном меню можно увидеть уже знакомые пункты **Build** (Собрать проект), **Rebuild** (Полностью собрать проект), **Add** (Добавить в проект новый или существующий файл), **Rename** (Переименовать проект) и **Remove** (Удалить проект из решения). Все эти команды будут выполняться в отношении выбранного проекта.

Если вы хотите сразу запустить проект на выполнение, то нажмите клавишу `<F5>` или выберите в главном меню окна команду **Debug | Start Debugging** (Отладка |

Запуск). В ответ на это проект будет запущен на выполнение с возможностью отладки — т. е. вы сможете устанавливать точки останова и выполнять код программы построчно. Если отладка не нужна, то нажимаем комбинацию клавиш `<Ctrl>+<F5>` или выбираем команду меню **Debug | Start Without Debugging** (Отладка | Запустить без отладки).

Даже самые простые проекты состоят из множества файлов, поэтому лучше проекты держать каждый в отдельном каталоге. Не пытайтесь объединять несколько проектов в один каталог — из-за этого могут возникнуть проблемы с поддержкой.

Среда разработки после компиляции также создает в каталоге проекта два каталога: `bin` и `obj`. В каталоге `obj` сохраняются временные файлы, которые используются для компиляции, а в каталоге `bin` — результат компиляции. По умолчанию применяются два режима компиляции: `Debug` и `Release`:

- ❑ в режиме `Debug` добавляется дополнительная информация, необходимая для тестирования и отладки. Такие файлы содержат много лишней информации, особенно если проект создан на C++, поэтому их только для тестирования и отладки и используют. Файлы, созданные в этом режиме компиляции, находятся в каталоге `bin\Debug`. Не поставляйте эти файлы заказчиком!
- ❑ `Release` — это режим чистой компиляции, когда в сборках нет ничего лишнего, и такие файлы поставляют заказчику или включают в установочные пакеты. Файлы, созданные в этом режиме компиляции, находятся в каталоге `bin\Release` вашего проекта.

На рис. 1.7 показан выпадающий список, с помощью которого можно менять режим компиляции.

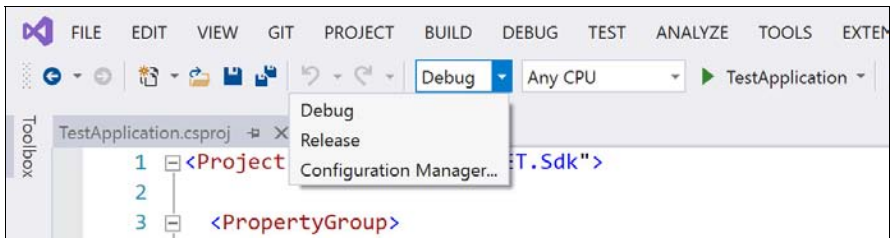


Рис. 1.7. Смена режима компиляции: `Debug` или `Release`

Впрочем, даже если компилировать проект в режиме `Release`, Visual Studio зачем-то помещает в каталог с результатом файлы с расширением `pdb`. Такие файлы создаются для каждого результирующего файла (библиотеки или исполняемого файла), причем и результирующий, и PDB-файл будут иметь одно имя, но разные расширения.

В PDB-файлах содержится служебная информация, упрощающая отладку, — она откроет пользователю слишком много лишней информации об устройстве кода. И если произойдет ошибка работы программы, а рядом с исполняемым файлом будет находиться его PDB-файл, то пользователю даже покажут, в какой строчке кода какого файла произошел сбой, чего пользователю совершенно не нужно знать.

Если же PDB-файла пользователю не давать, то в ошибке будет показана лишь поверхностная информация о сбое. Так что не распространяйте эти файлы вместе со своим приложением без особой надобности.

Впрочем, если вы разрабатываете веб-сайт, то подобный файл можно поместить на веб-сервер, если вы правильно обрабатываете ошибки. В этом случае в журнал ошибок будет записана подробная информация о каждом сбое, которая упростит поиск и исправление ошибки.

1.2.2. Работа с файлами

Чтобы начать редактирование файла, необходимо щелкнуть на нем двойным щелчком в панели **Solution Explorer**. В ответ на это действие в рабочей области окна появится вкладка с редактором соответствующего файла. Содержимое и вид окна сильно зависят от типа файла, который вы открыли.

Мы в основном будем работать с языком C#. В качестве расширения имени файлов для хранения исходного кода этот язык, как уже отмечалось ранее, использует комбинацию символов cs (C Sharp).

1.3. Простейший пример .NET-приложения

Большинство руководств и книг по программированию начинаются с описания простого примера, который чаще всего называют «Hello, World» («Здравствуй, мир»). Если ребенок, родившись на свет, издает крик, то первая программа будущего программиста в такой ситуации говорит: «Hello, World». Что, будем как все, — или назовем свою первую программу: «А вот и я» или «Не ждали»? Простите мне мой канадский юмор, который будет регулярно появляться на страницах книги...

Честно сказать, название не имеет особого значения, главное — показать простоту создания проекта и при этом рассмотреть основы. Начинать с чего-то более сложного и интересного нет смысла, потому что программирование — занятие не из простых, а необходимо рассказать очень многое.

Поэтому сейчас мы создадим самое простое приложение на C# для платформы .NET и с помощью этого примера разберемся в основах новой платформы, а потом уже начнем усложнять задачу.

1.3.1. Проект на языке C#

При рассмотрении возможностей Visual Studio (см. *разд. 1.2.1*) мы в качестве примера создали консольное приложение. Это как раз то, что нам нужно, и давайте продолжим разбираться с тем, что нами создано и как это работает.

И поскольку мы создали версию консольного приложения .NET 7, то наше приложение должно будет работать в Windows, Linux и macOS.

Как мы уже видели, в панели **Solution Explorer** у проекта **TestApplication** есть только один файл — **Program.cs**. Щелкните на нем двойным щелчком, и в рабочей области окна откроется редактор кода файла.

Если при создании проекта вы не установили флажок **Do not use top-level statements** (Не использовать код верхнего уровня), то в файле будет только одна строка с выводом текста в консоль. Иначе там будет кода чуть больше, но чтобы мы работали с одним и тем же кодом, удалите оттуда всё и наберите следующее:

```
using System;

namespace TestApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
        }
    }
}
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter1\TestApplication` сопровождающего книгу электронного архива (см. *приложение*).

1.3.2. Компиляция и запуск проекта на языке C#

Сначала скомпилируем проект и посмотрим на результат работы. Для компиляции необходимо выбрать в меню команду **Build | Build Solution** (Построить | Построить решение) или просто нажать комбинацию клавиш `<Ctrl>+<Shift>+<B>`. В панели **Output** (Вывод) отобразится информация о компиляции. Если этой панели у вас пока нет, то она появится после выбора указанной команды меню, или ее можно открыть, выбрав команду меню **View | Output**. Посмотрим на результат в панели вывода:

```
----- Завершено -----
Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped
Построено: 1 удачно, 0 с ошибками, 0 не требующих обновления, 0 пропущено
```

Я привел перевод только последней строки, потому что в ней пока кроется самое интересное. Здесь написано, что один проект скомпилирован удачно и ошибок нет. Еще бы — ведь в нашем проекте и кода почти нет.

Результирующий файл можно найти в каталоге вашего проекта — во вложенном в него каталоге `bin\ТекущаяКонфигурация\X.X` (*ТекущаяКонфигурация* здесь — это каталог `Release` или `Debug`), а `X.X` — это версия .NET (если создавать приложение .NET Framework, то последней части пути не будет). Получается, что исполняемый файл может находиться как в каталоге `bin\Debug`, так и в каталоге `bin\Release` — это зависит от типа компиляции. Напомню, что для смены режима компиляции можно использовать и выпадающий список на панели инструментов (см. рис. 1.7).

Обратите внимание, что в свойствах решения есть еще и свойство **Startup project** (Исполняемый проект). Если у вас в решение входят несколько проектов, то в этом

списке можно выбрать тот проект, который будет запускаться из среды разработки. Для запуска проекта выберите команду меню **Debug | Start** (Отладка | Выполнить).

Запускать наше приложение из среды разработки нет смысла, потому что окно консоли появится только на мгновение и вы не успеете увидеть результат, но попробуйте все же нажать клавишу <F5>.

Для того чтобы результат выполнения приложения все же оставался в окне, попробуем модифицировать код следующим образом:

```
using System;

class TestApplication
{
    public static void Main()
    {
        Console.WriteLine("Hello, World!!!");
        Console.ReadLine();
    }
}
```

В этом примере добавлена строка `Console.ReadLine()`, которая заставляет программу дожидаться, пока пользователь не нажмет клавишу <Enter>. Если теперь снова скомпилировать проект и запустить его на выполнение, то на фоне черного экрана командной строки вы сможете увидеть заветную надпись, которую видит большинство начинающих программистов: **Hello, World!!!** (кавычки, окружающие эту надпись в коде, на экран выведены не будут). Хотя я и обещал другую, более оригинальную надпись, не стоит все же выделяться из общей массы, так что давайте посмотрим именно на эти великие слова.

Итак, у нас получилось первое приложение, и вы можете считать, что первый шаг на долгом пути к программированию уже сделали. Шаги будут постепенными, и сейчас абсолютно не нужно понимать, что происходит в этом примере, — нам необходимо знать только две строки:

```
Console.WriteLine("Hello, World!!!");
Console.ReadLine();
```

Первая строка выводит в консоль текст, который указан в скобках. Кстати, текст должен быть размещен в двойных кавычках, если это именно *текст*, а не *переменная* (о переменных читайте в разд. 2.2 и 2.4). О строках мы тоже позднее поговорим отдельно, но я решил все же сделать это небольшое уточнение уже сейчас.

Вторая строка запрашивает у пользователя строку символов. Концом строки считается символ возврата каретки, который невидим, а чтобы ввести его с клавиатуры, мы нажимаем клавишу <Enter>. В любом текстовом редакторе, чтобы перейти на новую строку (завершить текущую), мы нажимаем эту клавишу, и точно так же и здесь.

Когда вы нажмете клавишу <Enter>, программа продолжит выполнение. Но у нас же больше ничего в коде нет — только символы фигурных скобок. А раз ничего

нет, значит, программа должна завершить работу. Вот и вся логика этого примера, и ее нам будет достаточно для следующей главы, в процессе чтения которой мы станем рассматривать не самые интересные, но очень необходимые на пути изучения C# примеры.

1.4. Компиляция приложений

В этом разделе мы чуть подробнее поговорим о компиляции C#-приложений. Для того чтобы создать сборку (мы уже создавали исполняемый файл для великого приложения «Hello, World»), необходимо нажать комбинацию клавиш <Ctrl>+<Shift>+, или клавишу <F6>, или выбрать в меню команду **Build | Build Solution**. Компиляция C#-кода в IL происходит достаточно быстро, если сравнивать этот процесс с компиляцией классических C++-приложений.

Но Visual Studio — это всего лишь удобная оболочка, в которой писать код — одно удовольствие. На самом же деле код можно писать в любой другой среде разработки или даже в Блокноте, а для компиляции использовать утилиту командной строки.

1.4.1. Компиляция в .NET Framework

Для компиляции под платформу .NET Framework используется программа компилятора `csc.exe` (от *англ.* C-Sharp Compiler), которую можно найти в каталоге `C:\Windows\Microsoft.NET\Framework\vX.X`, где `X.X` — версия .NET Framework. На моем компьютере этот компилятор находится в папке `C:\Windows\Microsoft.NET\Framework\v3.5\csc.exe` (у вас вложенная папка `v3.5` может иметь другое имя, которое зависит от версии .NET).

ПРИМЕЧАНИЕ

Раньше компилятор C# находился в папке `C:\Program Files\Microsoft.NET\SDK\vX.X\Bin\`. Сейчас там можно найти множество утилит .NET как командной строки, так и с визуальным интерфейсом.

Чтобы проще было работать с утилитой командной строки, путь к ней лучше добавить в переменные окружения. Для этого щелкните правой кнопкой мыши на ярлыке **Мой компьютер** (My Computer) на рабочем столе и выберите **Свойства** (Properties). Начиная с Windows 7, откроется окно, похожее на окно панели управления, в котором нужно выбрать пункт **Дополнительные параметры системы** (Advanced system settings). Далее нажмите кнопку **Переменные среды** (Environment Variables) и в системный параметр **Path** добавьте через точку с запятой путь к каталогу, где находится компилятор `csc`, который вы будете использовать.

Теперь попробуем скомпилировать пример «Hello, World», который мы написали ранее, а для этого в командной строке нужно выполнить следующую команду:

```
csc.exe /target:exe test.cs
```

В нашем случае `test.cs` — это файл с исходным кодом. Я просто его переименовал, чтобы не было никаких `Class1.cs`. Обратите внимание, что имя файла компилятора

указано полностью (с расширением), и именно так и должно быть, иначе он не запустится, если только путь к этому файлу в системном окружении вы не указали самым первым. Дело в том, что самым первым в системном окружении стоит путь к каталогу C:\Windows, а в этом каталоге уже есть каталог CSC, и именно его будет пытаться открыть система, если не указано расширение файла.

После имени файла идет ключ `/target` — он указывает на тип файла, который вы хотите получить в результате сборки. Нас интересует исполняемый файл, поэтому после ключа и двоеточия надо указать `exe`. Далее идет имя файла. Я не указываю здесь полный путь, потому что запускаю команду в том же каталоге, где находится файл `test.cs`, иначе пришлось бы указывать полный путь к файлу.

В результате компиляции мы получаем исполняемый файл `test.exe`. Запустите его и убедитесь, что он работает корректно. Обратите внимание на имя исполняемого файла. Если при компиляции из среды разработки оно соответствовало имени проекта, то здесь — имени файла. Это потому, что в командной строке мы компилируем не проект, а файл `test.cs`.

Конечно же, для больших проектов использовать утилиту командной строки проблематично и неудобно, поэтому и мы не станем этого делать. Но для компиляции небольшой программки из 10 строк, которая только показывает или изменяет что-то в системе, эта утилита вполне пригодна, и нет необходимости ставить тяжеловесный пакет Visual Studio. Где еще ее можно использовать? Вы можете создать собственную среду разработки и использовать `csc.exe` для компиляции проектов, но на самом деле я не вижу смысла это делать.

В составе .NET Framework есть еще одна утилита — `msbuild`, которая умеет компилировать целые проекты из командной строки. В качестве параметра утилита принимает имя файла проекта или решения, и она уже делает все то, что умеет делать Visual Studio во время компиляции в IDE.

С недавнего времени эту утилиту включили в состав Visual Studio, что не совсем понятно. Ведь вся сила `msbuild` заключается в том, что с ее помощью можно компилировать даже без установки VS, а теперь получается, что мне все равно придется использовать установочный пакет среды разработки.

Я по работе создаю достаточно большой сайт, а компилировать мы его должны на сервере, где нельзя установить Visual Studio. Однако, установив только .NET Framework, мы можем из командной строки собрать проект и выполнить скрипт для запуска обновленного кода на «боевых» серверах. Надо также отметить, что сервер, на котором мы собираем код, находится в другой стране, и к нему мы подключаемся удаленно. Так вот, используя `msbuild`, очень просто подключиться к серверу удаленно и все выполнить из командной строки.

1.4.2. Компиляция в .NET Core или .NET 5 и старше

Для компиляции приложения .NET Core и .NET 5 (скорее всего, и в более поздних версиях .NET) из командной строки нужно выполнить команду:

```
dotnet run
```

При создании приложения .NET Core формируется DLL — библиотечный файл, а не исполняемый, т. е. для нашего тестового приложения будет создан файл `TestApplication.dll`. Именно в этом файле хранится код приложения. В .NET рядом с этим библиотечным файлом будет создан еще и исполняемый файл `TestApplication.exe` — чтобы упростить запуск приложения в ОС Windows. Дело в том, что исполняемые файлы в разных ОС внутри выглядят по-разному, и нельзя создать один-единственный, который будет выполняться во всех ОС одинаково. Так что когда вы в Windows запускаете исполняемый файл `TestApplication.exe`, то в реальности он заставит фреймворк .NET выполнить промежуточный IL-код, который был скомпилирован в файле `TestApplication.dll`.

Для Linux или macOS у нас такого исполняемого файла нет. Тогда как же выполнить приложение? Для этого служит утилита `dotnet run`. Запустите командную строку — например, PowerShell. Перейдите в каталог проекта, где расположен файл `TestApplication.csproj`, — в моем случае это сделает команда:

```
cd C:\Users\fleno\source\repos\CSharpBook\Chapter1\TestApplication
```

и просто выполните команду:

```
dotnet run
```

Утилита `dotnet` запустит DLL-файл. Причем команда `dotnet run` в реальности еще и компилирует проект, так что отдавать команду `dotnet build` перед запуском приложения с помощью команды `dotnet run` не нужно.

Вы можете сказать, что это неудобно, и пользователи не будут счастливы, что им нужно помнить эту команду, но точно такая же проблема есть и у Java. Рассматриваемые нами платформы могут создать промежуточный IL-код, но кто-то должен инициировать выполнение кода...

Как уже было сказано, нельзя создать универсальный исполняемый файл, который будет запускаться в любой ОС, потому что у Linux и macOS другой формат исполняемого файла. Но платформа может сформировать исполняемый файл-заглушку, которая сделает все для нас, и для создания этого файла нужно опубликовать наше приложение:

```
dotnet publish --runtime win7-x64
```

Эта команда публикует приложение, которое создает заглушку для исполнения нашего приложения в Windows, начиная с версии 7, и на компьютере с 64-битным процессором. Но как мы уже видели, EXE-файл для Windows при компиляции в Visual Studio под Windows будет создан автоматически и без `dotnet publish`. Однако если исполняемый EXE-файл отсутствует, то его можно создать публикацией приложения с помощью следующих платформ для исполнения:

- ❑ `osx.10.11-x64` — для macOS;
- ❑ `ubuntu.16.04-x64` — для Linux.

1.5. Поставка сборок

Теперь немного поговорим о поставке скомпилированных приложений конечному пользователю. Платформу проектировали так, чтобы избежать двух проблем: сложной регистрации, присущей COM-объектам, и DLL hell¹, характерной для платформы старых приложений Windows (Win32), когда более старая версия библиотеки могла заменить более новую версию и привести к краху системы. Обе эти проблемы были решены весьма успешно, и теперь мы можем забыть о них как о пережитке прошлого.

Установка сборок на компьютер пользователя сводится к банальному копированию библиотек (DLL-файлов) и исполняемых файлов. По спецификации компании Microsoft для работы программы должно хватать простого копирования, и в большинстве случаев дело обстоит именно так. Но разработчики — вольные птицы, и они могут придумать какие-то привязки, которые все же потребуют инсталляции с первоначальной настройкой. Я не рекомендую делать что-то подобное без особой надобности — старайтесь организовать все так, чтобы процесс установки оставался максимально простым, хотя копирование может выполнять и программа установки.

Когда библиотека DLL хранится в том же каталоге, что и программа, то именно программа несет ответственность за целостность библиотеки и ее версию. Но, помимо этого, есть еще *разделяемые* библиотеки, которые устанавливаются в систему, чтобы любая программа могла использовать их ресурсы. Именно в таких случаях чаще всего возникает DLL hell. Допустим, пользователь установил программу А, в которой используется библиотека XXX.dll версии 5.0. Здесь XXX — это просто некое имя, которое не имеет ничего общего с «клубничкой», — с тем же успехом мы могли бы написать и YYY.dll. Вернемся к библиотеке. Допустим, что теперь пользователь ставит программу В, где тоже есть библиотека XXX.dll, но версии 1.0. Очень старая версия библиотеки несовместима с версией 5.0, и теперь при запуске программы А компьютер «превращается в чертенка» и начинается «ад». Как решить эту проблему? Чтобы сделать библиотеку разделяемой, вы можете поместить ее в каталог C:\Windows\System32 (по умолчанию этот каталог в системе скрыт), но от «ада» вам не уйти, и за версиями придется следить самому.

Проектировщики .NET поступили в этом случае гениально и предложили нам другой, более совершенный метод разделять библиотеки .NET между приложениями — глобальный кеш сборок GAC², который располагается в каталоге C:\Windows\assembly. Чтобы зарегистрировать библиотеку в кеше, нужно, чтобы у нее была указана версия и она была бы подписана ключом. Как это сделать — отдельная история, а сейчас мы пока рассматриваем только механизм защиты. Однако глобальный кеш сборок — это прерогатива .NET Framework. Пока что .NET 5/7 вроде бы не планирует реализовывать его и будет работать только с *приватными* сборками,

¹ DLL hell (DLL-кошмар, буквально: DLL-ад) — тупиковая ситуация, связанная с управлением динамическими библиотеками DLL в операционной системе.

² Global Assembly Cache — глобальный кеш сборок.

о которых далее. Поэтому все последующее изложение, в котором упоминается глобальный кеш сборок, относится к работе с .NET Framework.

Итак, посмотрите на кеш сборок — каталог `C:\Windows\assembly`. Здесь может присутствовать несколько файлов с одним и тем же именем, но разных версий — например, `Microsoft.Build.Engine`. Если запустить поиск по имени `Microsoft.Build.Engine.ni.dll`, вы найдете несколько файлов, но эти сборки будут иметь разные версии: 2.0 и 3.5.

За скопированную в кеш библиотеку с версией 2.0 система «отвечает головой», и если вы попытаетесь скопировать в кеш еще одну такую же библиотеку, но с версией 3.5, то старая версия не затрется, — обе версии будут сосуществовать без проблем. Таким образом, программы под версию 2.0 будут прекрасно видеть свою версию библиотеки, а программы под 3.5 — свою, и вы не получите краха системы из-за некорректной версии DLL.

Тут вы можете спросить: а что если я создам свою библиотеку и дам ей имя `Microsoft.Build.Engine` — смогу ли я затереть библиотеку `Microsoft` и устроить крах для DLL-файлов? По идее, это невозможно, если только разработчики не допустили ошибки в реализации. Каждая сборка подписывается так, что для подмены чужой библиотеки вам понадобится ключ, которым она подписана. И если в кеше окажутся две библиотеки с одинаковыми именами, но с разными подписями, они будут в кеше мирно сосуществовать, и программы станут видеть ту библиотеку (сборку), которую и задумывал разработчик. Когда программист указывает, что в его проекте нужна определенная сборка из GAC (именно из GAC, а не локально), то в метаданных сохраняется имя сборки, версия и ключ, и при запуске .NET Framework ищет именно эту библиотеку. Так что проблем не должно быть, и пока ни у кого не было, — значит, разработчики в `Microsoft` все сделали хорошо.

С помощью подписей легко бороться и с возможной подменой DLL-файла. Когда вы ссылаетесь на библиотеку, то можете указать, что вам нужна для работы именно эта версия, и VS запомнит версию и открытый ключ. Если что-то не будет совпадать, то приложение работать не станет, а значит, закрываются такие потенциальные проблемы, как вирусы, которые занимаются подменой файлов.

Сборки, которые находятся в том же каталоге, что и программа, называются *приватными* (private), потому что должны использоваться только этим приложением, и их подписывать необязательно. Сборки, зарегистрированные в GAC, называются *совместными* или *разделяемыми* (shared).

Теперь посмотрим, из чего состоит версия сборки, — ведь по ней система определяет, соответствует она требованиям программы или нет и можно ли ее использовать. Итак, версия состоит из четырех чисел:

- Major — основная версия;
- Minor — подверсия сборки;
- Build — номер полной компиляции (построения) в данной версии;
- Revision — номер ревизии для текущей компиляции.

Первые два числа характеризуют основную часть версии, а остальные два являются дополнительными. При запуске исполняемого файла, если ему нужен файл из GAC, система ищет такой файл, в котором основная часть версии строго совпадает. Если найдено несколько сборок с нужной основной версией, то система выбирает из них ту, что содержит максимальный номер подверсии. Поэтому, если вы только исправляете ошибку в сборке или делаете небольшую корректировку, изменяйте значение Build и Revision. Но если вы изменяете логику или наращиваете функциональность (а это может привести к несовместимости), то следует изменять основную версию или подверсию — это зависит от количества изменений и вашего настроения. В таком случае система сможет контролировать версии и обезопасит вас от проблем с файлами DLL.

Щелкните правой кнопкой мыши на имени проекта в панели **Solution Explorer** и выберите в контекстном меню команду **Properties**. На первой вкладке **Application** (Приложение) нажмите кнопку **Assembly Information** (Информация о сборке). Перед вами откроется окно (рис. 1.8), в котором можно указать информацию о сборке, в том числе и ее версию в полях **Assembly Version**.

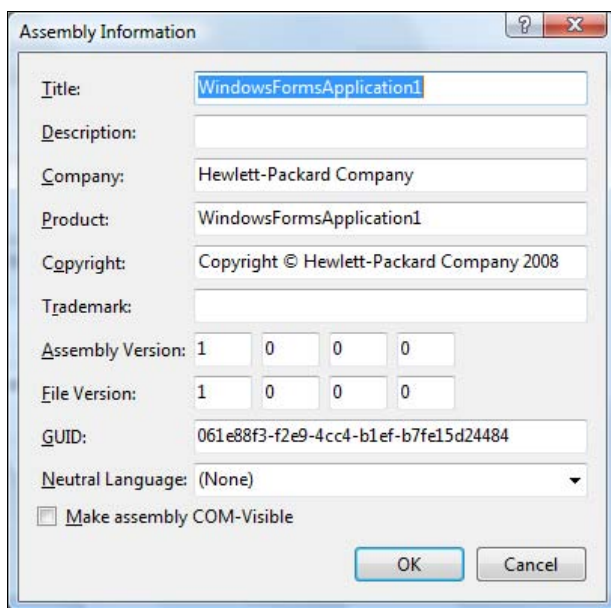


Рис. 1.8. Информация о сборке

Уже отмечалось, что исполняемые файлы, написанные под платформу .NET, не могут выполняться на процессоре, потому что не содержат машинного кода. Хотя нет, содержат, но только заглушку, которая сообщит пользователю о том, что нет виртуальной машины. Чтобы запустить программу для платформы .NET, у пользователя должна быть установлена ее нужная версия. В Windows 7/8/10/11 все необходимое уже есть, а для Windows XP, если кто-то ею еще пользуется, можно скачать и установить специальный пакет отдельно. Он распространяется компанией Microsoft

бесплатно и, по статистике, уже установлен на большинстве пользовательских компьютеров.

Пакет .NET, позволяющий выполнять .NET-приложения, вы можете скачать с сайта Microsoft по ссылке: <https://dotnet.microsoft.com/en-us/download>. Для .NET 2.0 этот файл занимает 23 Мбайт, а для .NET 3.0 — уже более 50 Мбайт. Ощущаете, как быстро и как сильно расширяется платформа? Но не это главное, главное — это качество, и пока оно соответствует ожиданиям большого количества разработчиков.

1.6. Формат исполняемого файла .NET

Для того чтобы вы лучше могли понимать работу .NET, давайте рассмотрим формат исполняемого файла этой платформы. Классические исполняемые файлы Win32 включали в себя:

- ❑ заголовок — описывает принадлежность исполняемого файла, основные характеристики и, самое главное, точку, с которой начинается выполнение программы;
- ❑ код — непосредственно байт-код программы, который будет выполняться;
- ❑ данные (ресурсы). В исполняемом файле могут храниться какие-то данные — например, строки или ресурсы.

В .NET-приложении в исполняемый файл добавили еще и метаданные.

C# является высокоуровневым языком, и он прячет от нас всю сложность машинного кода. Вы когда-нибудь просматривали Win32-программу с помощью дизассемблера или программировали на языке ассемблера? Если да, то должны представлять себе весь тот ужас, из которого состоит программа.

Приложения .NET не проще, если на них посмотреть через призму дизассемблера, только тут машинный код имеет другой вид. Чтобы увидеть, из чего состоит ваша программа, запустите утилиту `ildasm.exe`, которая входит в поставку .NET SDK. Если вы работаете со средой разработки Visual Studio .NET, то эту утилиту можно найти в каталоге `C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\Bin` (при использовании 7-й версии SDK).

Запустите эту утилиту, и перед вами откроется окно программы дизассемблера. Давайте загрузим в нее наш проект `TestApplication` и посмотрим, из чего он состоит. Для этого выбираем в главном меню утилиты команду **File | Open** и в открывшемся окне — исполняемый файл. В результате на экране будет отображена структура исполняемого файла.

В нашем исходном коде был только один метод — `Main()`, но, несмотря на это, в структуре можно увидеть еще метод `.ctor`. Мы его не описывали, но он создается автоматически.

Щелкните двойным щелчком на методе `Main()` и посмотрите на его код:

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
```



```
// Code size      11 (0xb)
.maxstack 1
IL_0000: ldstr "Hello World!!!"
IL_0005: call void [mscorlib]System.Console::WriteLine(string)
IL_000a: ret
} // end of method TestApplication::Main
```

Это и есть низкоуровневый код .NET-приложения, который является аналогом ассемблера для Win32-приложения. Помимо этого, вы можете увидеть в дереве структуры исполняемого файла пункт **MANIFEST**. Это манифест (текстовая информация или метаданные) исполняемого файла, о котором мы говорили ранее.

Как уже отмечалось, компиляторы .NET создают не байт-код, который мог бы выполняться в системе, а специальный IL-код. Исполняемый файл, скомпилированный с помощью любого компилятора .NET, будет состоять из стандартного PE-заголовка, который способен выполняться на Win32-системах, IL-кода и процедуры заглушки `CorExeMain()` из библиотеки `mscorlib.dll`. Когда вы запускаете программу, сначала идет стандартный заголовок, после чего управление передается заглушке. Только после этого начинается выполнение IL-кода из исполняемого файла.

Программы .NET Core и .NET не содержат Win32-кода. Вместо этого при публикации создается отдельный исполняемый файл, который и будет отвечать за инициацию выполнения сборки .NET Core/.NET.

Поскольку IL-код программы не является машинным и не может исполняться процессором, то он компилируется в машинный байт-код на лету с помощью специализированного JIT-компилятора. Конечно же, за счет компиляции на лету выполнение программы замедляется, но это только при первом запуске. Полученный машинный код сохраняется в специальном буфере на вашем компьютере и используется при последующих запусках программы.

1.7. Код верхнего уровня

В .NET 6 появилась возможность писать код верхнего уровня. Создайте еще один новый консольный проект, но на этапе выбора версии .NET (см. рис. 1.4) не устанавливайте флажок **Do not use top-level statements** (Не использовать код верхнего уровня).

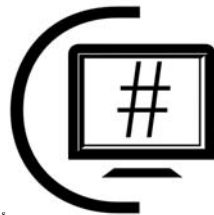
На этот раз среда разработки сгенерирует проект, у которого в файле `Program.cs` будут всего две строки: один комментарий и одна строка кода:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Несмотря на то что здесь только две строки, результат будет тот же самый. Просто теперь .NET может для файла `Program.cs` спрятать все сложности и сконцентрироваться только на самом важном.

В *разд. 1.3* мы создали более сложный скелет кода, чтобы увидеть полную версию. Но в следующей главе я буду использовать сокращенную версию кода из этого раздела.

ГЛАВА 2



ОСНОВЫ C#

Теоретических данных у нас теперь достаточно, и мы можем перейти к практической части и продолжить знакомство с языком C# на конкретных примерах. Именно практика позволяет лучше понять, что происходит и почему. Книжные картинки и текст — это хорошо, но когда вы сами сможете запустить программу и увидеть результат, то лучшего объяснения придумать просто невозможно.

Конечно, сразу же писать серьезные примеры мы не сможем, потому что программирование — весьма сложный процесс, для погружения в который нужно обладать достаточно глубокими базовыми знаниями, так что на начальных этапах нам помогут простые решения. Но о них — чуть позже. А начнем мы эту главу с рассказа о том, как создаются пояснения к коду (комментарии), и поговорим немного о типах данных и пространстве имен — т. е. зложим основы, которые понадобятся нам при рассмотрении последующих глав. Если вы знакомы с такими основами, то эту главу можете пропустить, но я рекомендовал бы вам читать все подряд.

2.1. Комментарии

Комментарии — это текст, который не влияет на код программы и не компилируется в выходной файл. А зачем тогда они нужны? С помощью комментариев я буду здесь вставлять в код пояснения, чтобы вам проще было с ним разбираться, поэтому мы и рассматриваем их первыми.

Некоторые используют комментарии для того, чтобы сделать код более читабельным. Хотя вопрос стиля выходит за рамки этой книги, я все же сделаю короткое замечание, что код должен быть все же читабельным и без комментариев.

Существуют два типа комментариев: однострочный и многострочный. *Однострочный* комментарий начинается с двух символов `//`. Все, что находится в этой же строке далее, — комментарий.

Следующий пример наглядно иллюстрирует сказанное:

```
// Объявление класса EasyCSharp
class EasyCSharp
```

```
{ // Начало
  // Функция Main
  public static void Main()
  {
    OutString()
  }
} // Конец
```

Символы комментария `//` не обязательно должны быть в начале строки. Все, что находится слева от них, воспринимается как код, а все, что находится справа до конца строки, — это комментарий, который игнорируется во время компиляции. Я предпочитаю ставить комментарии перед строкой кода, которую комментирую.

Многострочные комментарии в C# заключаются в символы `/*` и `*/`. Например:

```
/*
    Это многострочный
    комментарий.
*/
```

Если вы будете создавать многострочные комментарии в среде разработки Visual Studio, то она автоматически к каждой новой строке добавит в начало символ звездочки. Например:

```
/*
 * Это многострочный
 * комментарий.
*/
```

Звездочка не обязательна, но иногда помогает отличить строку комментария от строки кода. Во всяком случае такой комментарий выглядит элегантно и удобен для восприятия.

2.2. Переменная

Понятие *переменная* является одним из ключевых в программировании. Что это такое и для чего нужно? Любая программа работает с данными (числами, строками), которые могут вводиться пользователем или жестко прописываться в коде программы. Эти данные надо где-то хранить. Где? Постоянные данные хранятся в том или ином виде на жестком диске, а временные данные — в оперативной памяти компьютера.

Под *временными данными* я понимаю все, что необходимо программе для расчетов. Дело в том, что процессор умеет выполнять математические операции только над регистрами процессора (это как бы переменные внутри процессора) или в оперативной памяти. Поэтому, чтобы произвести расчеты над постоянными данными, их необходимо загрузить в оперативную память.

Допустим, мы загрузили данные в память, но как с ними теперь работать? В языке ассемблера для этого используются адреса памяти, где хранятся данные, а в высо-

коуровневых языках, к которым относится и C#, такие участки памяти имеют имена. Имя, которое служит для адресации памяти, и есть переменная. Имя проще запомнить и удобнее использовать, чем числовые адреса.

В .NET используется *общая система типов* (Common Type System, CTS). Почему именно «общая»? Дело в том, что приложения для этой платформы можно разрабатывать на разных языках. Раньше было очень сложно разрабатывать приложения сразу на нескольких языках, потому что у каждого была своя система типов, и хранение строки, скажем, в Delphi и в C++ происходило по-разному. Благодаря общности типов в .NET, на каком бы языке вы ни писали программу, типы будут одинаковые, и они одинаково будут храниться в памяти, а значит, станут одинаково интерпретироваться программой, и никаких проблем не возникнет.

В большинстве языков программирования выделяются два типа данных: простые (числа, строки, ...) и сложные (структуры, объекты, ...). В .NET и соответственно в C# такое разделение уже не столь четкое. Эта технология полностью объектная, и даже простые типы данных в ней являются по-своему *объектами* — просто обрабатываются платформой немного по-другому.

Полностью объектные типы данных уже пытались сделать не раз — например, в Java есть объекты даже для хранения чисел. Но это неудобно, поэтому для повышения скорости работы в Java применяются и простые переменные. При этом, когда нужно превратить простой тип в объект и обратно, производится упаковка и распаковка данных — т. е. конвертация простого типа данных в объект и из него.

В .NET типы можно разделить на значения (value type) и ссылочные (reference type). *Значения* — это как раз и есть простые типы данных. Если вам нужно сохранить в памяти число, то нет смысла создавать для этого объект, а достаточно просто воспользоваться типом значений.

Ссылочные типы — это объекты, а имя переменной — ссылка на объект. Обратите внимание, что это *ссылка*, а не указатель. Если вы слышали об указателях и об их уязвимости — не пугайтесь, в C# в основном применяются ссылки, хотя и указатели тоже возможны. Непонятно, что такое объект? Ничего страшного, пока для нас достаточно понимать, что есть простые типы данных: символы, числа (это легко представить) и нечто более сложное, называемое *объектами*.

В первой колонке табл. 2.1 представлены имена .NET типов данных. Что-то из этого классы, а что-то — структуры. Что такое *классы*, мы узнаем в *главе 3*, а структуры будут рассмотрены в *главе 4*. Я буду больше пользоваться более простыми вариантами имен, которые показаны во второй колонке, — т. е. *псевдонимами*.

Таблица 2.1. Основные типы данных общей системы типов (CTS)

Тип	Псевдоним	Описание
String	string	Строка
SByte	sbyte	8-разрядное число со знаком. Возможные значения от -128 до 127

Таблица 2.1 (окончание)

Тип	Псевдоним	Описание
Byte	byte	8-разрядное число без знака. Значения от 0 до 255
Int16	short	16-разрядное число со знаком. Возможные значения от −32 768 до 32 767
UInt16	ushort	16-разрядное число без знака. Значения от 0 до 65 535
Int32	int	32-разрядное число со знаком. Возможные значения от −2 147 483 648 до 2 147 483 647
UInt32	uint	32-разрядное число без знака. Значения от 0 до 4 294 967 295
Int64	long	64-разрядное число со знаком. Возможные значения от −9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
UInt64	ulong	64-разрядное число без знака. Значения от 0 до 18 446 744 073 709 551 615
Decimal	decimal	128-разрядное число
Char	char	16-разрядный символ
Single	float	32-разрядное число с плавающей точкой стандарта IEEE
Double	double	64-разрядное число с плавающей точкой
Boolean	bool	Булево значение

Основные типы описаны в пространстве имен `System`, поэтому для указания полного имени нужно добавлять `System` перед типом данных, например:

```
System.Int32
```

Чтобы не писать такого длинного имени в начале файла, очень часто его записывают так:

```
using System;
```

Это подключение *пространства имен*, о котором мы поговорим в *разд. 2.3*.

Самое интересное, что при использовании псевдонимов пространство имен не нужно. Даже если у вас не подключено ни одно пространство имен, следующая запись будет вполне корректной:

```
int i;
```

Таким образом, использование псевдонимов не требует подключения никаких пространств имен. Они существуют всегда. Я не знаю, с чем это связано...

Обратите внимание, что почти все типы данных имеют строго определенный размер, т. е. представляют собой как бы типы-значения. Когда вы объявляете переменную какого-либо типа, система знает, сколько нужно выделить памяти для хранения данных указанного типа. Для простых переменных память выделяется автома-

тически — и не только в .NET, но и в классических приложениях для платформы Win32. Нам не нужно заботиться о выделении или уничтожении памяти — все эти заботы берет на себя система: мы только объявляем переменную и работаем с ней.

Исключением является *строка*, длина которой может варьироваться. Для строки как для ссылочного объекта будет выделяться необходимая память, а в переменной сохраняться только ссылка.

2.3. Именованние элементов кода

Я всегда стараюсь уделить немного внимания именованию, потому что это — основа любой программы. От того, как будут выбираться имена, зависит читабельность кода приложения, а чем понятнее код, тем проще с ним работать — писать его и сопровождать.

Давайте сначала поговорим о *пространстве имен*. На мой взгляд, это достаточно удобное решение, которое сейчас присутствует во многих языках программирования.

Пространство имен — это определенная область, внутри которой все имена должны быть уникальными. И именно благодаря использованию пространства имен уникальность переменных должна обеспечиваться только внутри — т. е. в разных модулях (файлах) могут иметься переменные с одинаковыми именами.

Две разные компании могут создать код для работы с Сетью с одним и тем же именем — `Browser`. Но одна компания положит свой код в пространство имен `Chrome`, а другая — в пространство имен `Firefox`. Чтобы вызвать нужный, мы должны будем указать имя пространства имен и через точку тот `Browser`, который нам необходим:

```
Chrome.Browser
```

Это как названия улиц. Вот, например, вы видите в каком-либо тексте название улицы — Королева, а где она находится? Очень сложно с ходу дать правильный ответ, потому что улицу с таким названием можно найти в разных городах, так что как раз название города может выступать здесь в качестве пространства имен. А вот если мы напишем `Москва.Королева`, то теперь точно будем знать, какую улицу какого города мы имеем в виду.

В .NET все: классы, структуры и т. п. — разбито по пространствам имен, что позволяет избавиться от возможных конфликтов в именовании и в то же время использовать одинаковые имена в разных пространствах.

Пространство имен определяется с помощью ключевого слова `namespace` следующим образом:

```
namespace Имя
{
    Определение типов
}
```

Так можно определить собственное пространство имен, которое действует между фигурными скобками, внутри которых можно объявлять свои типы. Имя пространства имен может состоять из нескольких частей, разделенных точками. В качестве первой части рекомендуется указывать название фирмы, в которой вы работаете. Если вы программист-одиночка, то можете написать свое собственное имя. Следующий пример показывает, как объявить пространство имен с именем `TextNamespace` для организации `MyCompany`:

```
namespace MyCompany.TextNamespace
{
    Определение типов, классов, переменных
}
```

В C# 10 появился еще один способ объявить пространство имен — задать его в начале файла и сразу же поставить точку с запятой без указания фигурных скобок:

```
namespace MyCompany.TextNamespace;
Определение типов, классов, переменных
```

Этот вариант объявления пространства имен идентичен предыдущему, просто работает только, начиная с C# 10.

Если не указать никакого пространства имен, то имена будут попадать в глобальное пространство имен, и в этом случае может возникнуть конфликт из-за того, что два разных файла могут использовать одно и то же имя, поэтому рекомендуется не игнорировать пространства имен.

Когда вы разрабатываете небольшую программу, то достаточно просто контролировать именование и не допускать конфликтов. Но если проект большой и ваш код используется другими разработчиками, то без пространств имен не обойтись.

В C# нужно, чтобы даже маленькие проекты были заключены в какое-то пространство имен, и в одном проекте может быть использовано несколько пространств. Это всего лишь хороший, но нужный метод логической группировки данных.

В своих проектах я для каждой сборки создаю свое пространство.

Как хорошо выбрать пространство? Лично я предпочитаю использовать принцип *компания.проект.тематика*. У меня есть свой домен **www.profwebdev.com**, и для своих проектов я часто использую имя `profwebdev`:

```
ProfWebDev.Имя_Программы
```

или

```
ProfWebDev.Имя_Библиотеки
```

Для доступа к типам, объявленным внутри определенного пространства имен, нужно писать так:

```
Имя_Пространства_Имен.Переменная
```

Мы изучали пока только переменные, но сюда можно отнести еще и классы, которые мы будем подробно рассматривать в *главе 3*.

Для доступа к переменной можно использовать и сокращенный вариант — только имя переменной, но для этого должно быть выполнено одно из двух условий:

- мы должны находиться в том же самом пространстве имен. То есть, если переменная объявлена внутри фигурных скобок пространства имен и используется в них же, имя пространства писать не обязательно;
- мы должны подключить пространство имен с помощью оператора `using`.

Вспомните пример из *разд. 1.3.1*, где в самом начале подключается пространство имен `System` с помощью строки:

```
using System;
```

В этом пространстве описаны все основные типы данных, в том числе и те, что мы рассматривали в табл. 2.1, и все основные инструменты и функции работы с системой. Наверное, все приложения .NET обязательно подключают это пространство имен, иначе даже типы данных пришлось бы писать в полном варианте. Например, целое число придется писать как `System.Int32`, но если в начале модуля вставить строку `using System;`, то достаточно написать только `Int32` — без префикса `System`.

Кстати, если не подключить пространство имен `System`, то доступ к консоли для вывода текста тоже придется писать в полном виде, — ведь консоль также спрятана в пространстве имен `System`, и полный формат команды вывода на экран будет таким:

```
System.Console.WriteLine("Hello World!!!");
```

Но поскольку у нас пространство имен `System` уже подключено, то его указывать в начале команды необязательно.

Теперь поговорим о том, как правильно выбирать имена для переменных, методов и классов. Когда с программой работает множество человек или код слишком большой, то очень важно *правильно* именовать переменные. Давным-давно общепринятым стандартом было использование в начале имени чего-либо, указывающего на тип переменной. Если переменная должна хранить число, то перед именем ставили букву *i*, которая означала *integer*, или число. Для строк перед переменной ставили букву *s* (*string*, строка). Это старый подход, и сейчас он уже не рекомендуется к использованию.

Имя переменной должно быть таким, чтобы оно отражало смысл ее предназначения. Ни в коем случае не объявляйте переменную из одной или двух бессмысленных букв, потому что уже через месяц вы не сможете вспомнить, зачем нужна такая переменная и для чего вы ее использовали.

В моих реальных программах (не в учебных примерах) одной буквой называются только переменные с именем *i* или *j*, которые предназначены для счетчиков в циклах. Их назначение заранее предопределено, а для других целей переменные с такими именами не используются. Никаких префиксов у имен переменных не должно быть — только собственно имя, которое дает вам четко понять, что в этой переменной хранится. Рассмотрим несколько примеров:

- ❑ `sum` — скорее всего, в этой переменной будет какая-то сумма. Нам должно быть все равно, какого типа сумма: целое число или с плавающей точкой, главное — смысл значения;
- ❑ `filename` — явно имя файла и, скорее всего, строка, раз это имя;
- ❑ `productPrice` — цена товара.

Если вы знаете английский, то, наверное, обратили внимание, что все переменные — это имена существительные. Переменные — это данные, они не выполняют каких бы то ни было действий.

В этой книге для именования переменных и прочих элементов в простых примерах я буду иногда отходить от рекомендованных здесь правил. Но когда вы станете создавать свое приложение, то старайтесь следовать удобному для себя стилю с самого начала. В будущем вы увидите все преимущества правильного структурированного кода и правильного именования переменных.

Когда вы пишете программу, то помните, что на этапе разработки соотношение затраченных усилий и цены — минимальное. Не жалейте времени на правильное оформление, чтобы не ухудшить свое положение во время поддержки программы, а за нее платят намного меньше. При поддержке плохого кода вы будете тратить слишком много времени на вспоминание того, что делали год или два назад. В определенный момент может даже случиться так, что написание нового продукта с нуля обойдется дешевле поддержки старого.

ПРИМЕЧАНИЕ

Начиная с Visual Studio 2008, можно именовать переменные и другие пользовательские типы, используя символы кириллицы (русскими именами). Как я к этому отношусь? Я программист старой закалки и пишу код еще с тех времен, когда компиляторы понимали только английский язык. Поэтому я привык давать переменным англоязычные имена. Я не против русских имен — возможно, это и удобно, но просто у меня за многие годы выработалась привычка, от которой сложно избавиться. Прошу прощения, но большинство переменных я буду именовать по старинке — на английском.

2.4. Работа с переменными

Мы определились с тем, что такое *переменные*, но самое интересное заключается в том, как их использовать. Давайте посмотрим, как можно объявить переменную типа `int`, а затем присвоить ей значение:

```
int i;    // объявление переменной i
i = 10;   // присваивание переменной значения 10
Console.WriteLine(i);
```

Переменные в C# объявляются в любом месте кода. От того, где объявлена переменная, зависит, где она будет видна, но об этом мы еще поговорим далее. Сейчас же нам достаточно знать, что объявление переменной должно быть выполнено до ее применения и желательно как можно ближе к первому использованию.

В приведенном примере объявляется переменная `i`, которая будет иметь тип `int`, т. е. хранить целочисленное значение (вспоминаем табл. 2.1, где были приведены простые типы, в том числе и `int`). Объявление переменных происходит в виде:

```
ТИП ИМЯ;
```

Сначала пишем тип переменной, а затем через пробел имя переменной. Чтобы присвоить переменной значение, необходимо использовать конструкцию:

```
ИМЯ = значение;
```

В нашем примере мы присваиваем переменной `i` значение 10:

```
i = 10;
```

После этого участок памяти, на который указывает переменная `i`, будет содержать значение 10. Когда мы объявляем такие простые переменные, то .NET автоматически выделяет память в соответствии с их размером, и нам не нужно заботиться, где и как это происходит. Для более опытных программистов, конечно же, желательно иметь понимание процессов управления памятью, а пока можно обойтись без этого.

Вот тут нужно сделать небольшое отступление. Каждый оператор должен заканчиваться точкой с запятой. Это необходимо, чтобы компилятор знал, где заканчивается один оператор и начинается другой. Дело в том, что операторы необязательно должны вписываться в одну строку, и необязательно писать только по одному оператору в строку. Вы легко можете написать:

```
int
    i;
i
=
    10; Console.WriteLine(i);
```

В первых двух строках объявляется переменная `i`. В следующих трех — переменной устанавливается значение. При этом в последней строке есть еще вывод содержимого переменной на экран. Этот код вполне корректен, потому что все разделено точками с запятой — именно по ним компилятор будет отделять отдельные операции.

Несмотря на то что этот код вполне корректен, никогда не пишите так. Если строка кода слишком большая, то ее можно разбить на части, но писать в одной строке два действия не нужно. Строка получится перегруженной информацией и сложной для чтения.

Для объявления переменной `i` мы использовали псевдоним `int`. Полное название типа — `System.Int32`, и никто не запрещает использовать его:

```
System.Int32 i;
i = 10;
Console.WriteLine(i);
```

Результат будет тем же самым. Но большинству программистов просто лень писать такое длинное определение типа. Я думаю, что вы тоже будете лениться — ведь три буквы `int` намного проще написать, чем длинное определение `System.Int32`.

ВНИМАНИЕ!

Язык C# чувствителен к регистру букв. Это значит, что переменная `i` и переменная `I` — совершенно разные переменные. То же самое и с операторами языка — если тип называется `int`, то нельзя его писать большими буквами `INT` или с большой буквы `Int`. Вы должны четко соблюдать регистр букв, иначе компилятор выдаст ошибку.

Если нужно объявить несколько переменных одного и того же типа, то их можно записать через запятую, например:

```
int i, j;
```

Здесь объявляются две целочисленные переменные с именами `i` и `j`.

Теперь посмотрим, как определяются строки. Следующий пример объявляет строку `s`, присваивает ей значение и выводит содержимое на экран:

```
System.String s;  
s = "Hello";  
Console.WriteLine(s);
```

Объявление строк происходит так же, как и целочисленных переменных. Разница в присвоении значения. Значение строки должно быть заключено в двойные кавычки, которые указывают на то, что это текст.

Присваивать значение переменной можно сразу же во время объявления, например:

```
int i = 10;
```

В этом примере объявляется переменная и тут же ей назначается значение.

Ставить пробелы вокруг знака равенства не обязательно, и строку можно было бы написать так:

```
int i=10;
```

Но этот код, опять же, выглядит не слишком красиво. Помните, что красота — это не только баловство, это еще и удобство. Кто-то очень умный как-то сказал, что мы пишем код только один раз, а читаем потом многократно. Возможно, я сейчас привел это высказывание не дословно, но смысл его точно был таким.

Из личного опыта могу сказать, что большинству случаев мне приходится работать с целыми числами и типом данных `int`. Но я также очень часто пишу экономические и бухгалтерские программы, а вот тут уже нужна более высокая точность данных, где участвует запятая, — задействуются вещественные или дробные числа. Для хранения дробных чисел в .NET обычно используют тип данных `double`, потому что его размера вполне достаточно. В коде программы дробная часть — вне зависимости от региональных настроек — записывается через точку:

```
double d = 1.2;
```

В этой строке кода объявляется переменная `d`, которой сразу же присваивается значение одна целая и две десятых.

А вот когда вы запустите программу и попросите пользователя ввести данные, то он должен будет вводить их в соответствии с установленными в операционной системе или в браузере региональными настройками.

Что произойдет, если попытаться присвоить значение одной переменной другой переменной? Мы уже знаем, что переменная — это имя какого-то участка памяти. Когда мы присваиваем значение одной переменной другой переменной, будут ли обе переменные указывать на один участок памяти? Это зависит от ряда условий. Например, для простых типов — таких как числа, в процессе присвоения копируется значение: каждая переменная является именем своего участка памяти, и в процессе присвоения одно значение из своего участка памяти копируется в участок памяти другого. Но это не касается ссылочных типов, где копируется ссылка, — в этом случае обе переменные начинают ссылаться на один и тот же объект в памяти.

Переменные мы будем использовать очень часто и в большинстве примеров, поэтому тема работы с ними останется с вами надолго.

Рассмотрим простую, но очень полезную задачку на работу с переменными. Допустим, что у нас есть две переменные:

```
string firstname = "Фленов";  
string lastname = " Михаил";
```

Здесь кто-то явно ошибся и в переменную для имени (`firstname`) поместил фамилию ("Фленов"), а в переменную для фамилии попало имя. Как поменять эти две переменные местами? Если мы попытаемся скопировать фамилию в имя:

```
firstname = lastname;
```

то тогда в обеих переменных будет Михаил. Мы просто затерли имя и потеряли данные. Вместо этого надо завести еще одну переменную:

```
// скопировали Фленов из firstname в переменную temp  
string temp = firstname; // в temp теперь Фленов  
firstname = lastname; // копируем Михаил в firstname  
lastname = temp; // копируем из переменной temp фамилию
```

Таким образом мы за счет дополнительной переменной поменяли значения в двух ячейках памяти местами.

Когда нужно объявить несколько переменных одного типа, то это можно сделать через запятую:

```
int a = 1, b = 2, c = 3;
```

Здесь через запятую объявлены три переменные типа `int` и им всем назначены значения.

2.4.1. Строки и символы

О *строках* мы уже говорили в *разд. 1.3.2*, и там мы узнали, что они заключаются в двойные кавычки:

```
string str = "Это строка";
```

Строки — это не совсем простые переменные. Вспомните табл. 2.1, где были указаны простые типы данных в .NET и их размеры. Когда вы объявляете переменную,

то система сразу знает, сколько памяти нужно выделить для хранения значения. А как быть со строками? Каждый символ в строке занимает два байта (в .NET для хранения строк используется кодировка Unicode, но они могут по мере надобности, преобразовываться и в другие кодировки), однако количество символов в строке далеко не всегда возможно узнать заранее. Как же тогда быть?

В Win32 и классических приложениях на языке C/C++ программист должен был перед использованием строки выделить для нее память. Чаще всего это делалось одним из двух способов: с помощью специализированных функций или с помощью объявления *массива* из символов (о массивах читайте в *разд. 2.4.2*). Но оба способа не идеальны и при неаккуратном использовании приводили к переполнению буфера или выходу за пределы выделенной памяти. При этом проникший в систему хакер мог изменять произвольные участки памяти, приводить к краху системы или даже взламывать компьютер.

Новую платформу .NET разрабатывали максимально безопасной, поэтому безопасность строк стояла на первом месте. Чтобы не «изобретать велосипед» и не ошибиться, разработчики посмотрели, что уже существует в мире, и взяли из него самое лучшее. Мне кажется, что за основу была взята работа со строками в Java, где строки создаются только один раз и система сама выделяет память для их хранения.

Рассмотрим пример кода:

```
string str = "Это строка";
```

Здесь переменной *str* присваивается текст, а .NET Framework во время выполнения может без проблем подсчитать количество символов в строке, выделить память для хранения текста и сохранить там данные.

А что если переменной присваивается текст, который вводится пользователем в каком-то окне? В этом случае система должна сначала получить вводимую информацию, подсчитать ее размер, выделить необходимую память и сохранить туда введенную информацию.

Теперь посмотрим на следующие три строки кода:

```
string str;  
str = "Это строка";  
str = "Это еще одна строка!";
```

В первой строке мы просто объявляем переменную с именем *str*. Мы еще не присвоили ей никакого значения, а значит, память для нее не будет выделена, и *str* останется просто непроинициализированной. Если попытаться обратиться к такой переменной, произойдет ошибка доступа к непроинициализированной переменной.

Во второй строке кода переменной присваивается текст. Именно в этот момент под переменную *str* будет выделена память, и текст будет сохранен в эту память. Тут все понятно и легко, потому что система подсчитывает количество символов (их 10) и выделяет память для них.

Следующая строка кода присваивает все той же переменной `str` новый текст, в котором теперь 20 символов. Но система уже выделила для `str` память, и там хватает места только для половины символов, куда же девать остальные? В .NET, чтобы избежать таких ситуаций, изменять строки запрещено, и при каждой попытке их изменения просто создается новый экземпляр. Что это значит? Несмотря на то что `str` уже проинициализирована и содержит значение, память старого значения будет уничтожена, и вместо нее будет создана новая переменная с необходимым количеством памяти. Чувствуете мощь?

Мощь — это хорошо, но она бьет по скорости выполнения кода. Если при каждом изменении уничтожать старую память и выделять новую, то ресурсы процессора станут расходоваться на дополнительные операции обеспечения безопасности, а ведь часто можно обойтись без пересоздания переменной. Тем не менее безопасность, во-первых, все же важнее скорости, а во-вторых, есть методы работы со строками, требующие частого их изменения, но выполняющиеся очень быстро, и о них мы еще поговорим.

Обратите внимание, что система сама выделяет память для хранения нужной строки требуемого объема. В классических приложениях Win32 программистам очень часто приходилось четко говорить: сколько памяти выделить и когда уничтожить выделенную память. В .NET в этом нет необходимости — платформа сама берет на себя все заботы по уничтожению любой выделенной памяти.

Помимо строк, в .NET есть еще один тип данных, который может хранить *символ*. Да, именно один символ, и это тип `char`:

```
char ch = 'f';
```

Здесь показано, как объявить переменную `ch` типа `char` и присвоить ей значение — символ `'f'`. Обратите внимание, что строки в C# обрамляются двойными кавычками, а одиночный символ типа `char` — одинарными.

В переменную типа `char` можно поместить один и только один символ. Нельзя присвоить два символа — следующий код:

```
char ch = 'aб';
```

завершится ошибкой «Too many characters» (слишком много символов).

Нельзя указывать пустой символ:

```
char ch = '';
```

Вы получите ошибку «Empty character illegal» (некорректный пустой символ).

А вот строку `string` можно сделать пустой:

```
string ch = "";
```

Если же вы присваиваете один символ строке, то его нужно обрамлять двойными кавычками:

```
string ch = "f";
```

Да, именно так: несмотря на то, что мы в переменной `ch` сохраняем только один символ, его нужно обрамлять двойными кавычками, потому что переменная `ch` имеет тип `string`.

2.4.2. Массивы

Уметь хранить в памяти одно значение любого типа — это хорошо, но может возникнуть необходимость хранить в памяти группу значений одинакового типа. Допустим, что нужно сохранить где-то несколько чисел — например: 10, 50 и 40. Пока не будем вдаваться в подробности, зачем это нужно и что означают числа, а лишь представим, что это просто необходимо.

Для хранения нескольких чисел можно создать три переменные, а можно создать только одну переменную, но в виде *массива* из трех целых чисел, и обращаться к значениям массива по индексу. Тут можно подумать, что проще завести все же три переменные и не думать о каких-то массивах. Но что вы станете делать, когда нужно будет сохранить 100 различных значений? Объявлять 100 переменных — это катастрофа. Проще шубу заправить в брюки, чем реализовать этот код, хотя заправлять шубу в брюки — такая же глупость, как и объявлять 100 переменных.

Итак, как же мы можем объявить массив определенного типа данных? Для этого после типа данных нужно указать квадратные скобки. Например, если простую числовую переменную мы объявляем так:

```
int переменная;
```

то массив из чисел объявляется так:

```
int[] переменная;
```

Теперь у нас есть переменная, и мы могли бы ее использовать, если бы не одно очень большое и жирное **НО** — переменная не проинициализирована. Как мы уже знаем, простые типы данных имеют определенный размер, и система может выделить память для их хранения автоматически, а тут перед нами массив, и мы даже не знаем, какой у него размер (сколько элементов он будет хранить). Это значит, что система при всем желании не может знать, сколько памяти нужно зарезервировать под данные массива.

Чтобы проинициализировать переменную массива (непосредственно выделить память), используется оператор `new`, а в общем виде инициализация выглядит так:

```
переменная = new тип[количество элементов];
```

Допустим, нам нужен массив из трех чисел — его можно объявить и проинициализировать следующим образом:

```
int[] intArray;  
intArray = new int[3];
```

В первой строке мы объявляем переменную, а во второй присваиваем ей результат инициализации для трех элементов. А ведь то же самое можно сделать в одной строке — сразу же объявить переменную и тут же присвоить ей результат инициализации, как мы уже делали это с простыми типами. Следующая строка кода объявляет и инициализирует переменную в одной строке:

```
int[] intArray = new int[3];
```

Только с простыми типами мы не писали слово `new`. Как мы уже знаем, число имеет четкий размер, и для хранения числа не нужно выделять память — достаточно объявить переменную, и память уже будет выделена. Массив же может хранить большое количество значений, и поэтому инициализировать его нужно с помощью `new`, и в этот момент для хранения указанного количества элементов указанного типа будет выделена необходимая память. В приведенном примере память будет выделена для хранения трех чисел.

Теперь у нас есть одна переменная, и мы можем хранить в ней три разных значения одновременно. Но как к ним обращаться? Очень просто — после имени переменной в квадратных скобках пишем индекс элемента, к которому нужно обратиться. Следующий пример присваивает элементу с индексом 1 значение 50:

```
intArray[1] = 50;
```

И вот тут самое интересное и очень важное — нумерация элементов в массиве. Элементы массивов в C#, как и в большинстве других языков программирования, нумеруются с нуля. Это значит, что если мы создали массив для трех элементов, то их индексы будут 0, 1 и 2, а не 1, 2 и 3. Обязательно запомните это, иначе будете часто видеть сообщение об ошибке выхода за пределы массива. Ничего критического для системы вы сделать не сможете, но вот ваша программа будет завершать работу аварийно.

В следующем примере я сохраняю в трех элементах массива три значения: 10, 50 и 40:

```
intArray[0] = 10;  
intArray[1] = 50;  
intArray[2] = 40;
```

Раз уж мы заговорили о пределах массива, сразу же скажу об их безопасности. В Win32-приложениях выход за пределы массива, как и выход за пределы строки, был очень опасен и вел к тем же самым последствиям. При выходе за пределы размера строки ошибка не генерировалась, а данные за этими пределами изменялись, что могло разрушить важные участки данных, попавших за пределы области памяти строки.

На самом деле, строки в Win32 — это разновидность массива, просто это массив одиночных символов, т. е. в C# он имел бы следующий вид:

```
char[] строка;
```

Именно так можно объявить массив символов и в каждый элемент массива поместить соответствующую букву слова.

Поскольку выход за пределы массива опасен для программы или даже для ОС, платформа .NET защищает нас от выхода за пределы массива, и вы можете обращаться только к выделенной памяти. Да, тут мы теряем в гибкости, но зато выигрываем в безопасности, — при попытке обратиться к элементу за пределами массива будет сгенерирована ошибка.

Теперь посмотрим на небольшой пример, который объявляет и инициализирует массив из трех чисел, а потом выводит содержимое массива на экран:


```
int[] intArray = new int[3];

intArray[0] = 10;
intArray[1] = 50;
intArray[2] = 40;
intArray[3] = 40;    // ошибка, мы ведь выделили массив из трех элементов

Console.WriteLine(intArray[0]);
Console.WriteLine(intArray[1]);
Console.WriteLine(intArray[2]);
```

Инициализация массива с помощью оператора `new` очень удобна, когда элементы заполняются расчетными данными или каким-то другим способом. Когда же количество элементов и их значения известны заранее, такая инициализация свое удобство теряет. Например, для создания массива с названиями дней недели придется писать как минимум 8 строк: одну — для объявления и инициализации и 7 строк — для заполнения массива значениями. Это нудно и скучно, поэтому для случаев, когда значения известны заранее, придумали другой способ инициализации:

```
Переменная = { Значения, указанные через запятую };
```

Тут не нужен оператор `new` и не надо указывать размер массива: система подсчитает количество элементов в фигурных скобках и выделит соответствующее количество памяти. Например, следующий код показывает, как создать массив с названиями дней недели за один оператор (я просто записал его в две строки для удобства), после чего на экран выводится третий элемент массива (который имеет индекс 2 — надеюсь, вы не забыли, что элементы нумеруются с нуля):

```
string[] weekDays = { "Понедельник", "Вторник",
    "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье" };
Console.WriteLine(weekDays[2]);
```

Пока что этой информации о массивах вам достаточно. Постепенно, используя массивы, мы узнаем о них больше.

Массивы не ограничены только одним измерением. Если вам нужно сохранить таблицу данных, вы можете создать двумерный массив:

```
int[,] myArray;
myArray = new int[3,3];
```

Здесь объявляется двумерный массив с именем `myArray`. Размерность массива легко подсчитать, прибавив единицу к количеству запятых внутри квадратных скобок в первой строке. Если запятых нет, то к нулю прибавляем 1, а это значит, что по умолчанию — без запятых — создается одномерный массив.

Во второй строке происходит инициализация массива. Обратите внимание, что в квадратных скобках через запятую указана величина каждой размерности. Для одномерного массива мы указывали только одно число, а тут нужно указывать две размерности: *X* и *Y*. В нашем случае система выделит в памяти таблицу для хранения целых чисел размером 3×3 элемента.

Следующий пример показывает, как можно объявить и тут же создать трехмерный массив данных:

```
int[, ,] myArray = new int[6, 6, 5];
```

Доступ к элементам многомерного массива происходит почти так же, как и к одномерным, — просто в квадратных скобках нужно через запятую указать индексы каждой размерности элемента, к которому вы хотите получить доступ. Следующая строка изменяет значение элемента двумерного массива с индексом (1, 1):

```
myArray[1, 1] = 10;
```

Если вы используете трехмерный массив, то в квадратных скобках придется указать значения всех трех размерностей.

2.4.3. Перечисления

Следующее понятие, которое мы рассмотрим, — это *перечисления*: `enum`. Перечисления — не совсем тип данных... Я бы сказал, что это способ создания собственных удобных типов данных для наборов значений небольшого размера. Если сказанное непонятно, не пытайтесь сходу вникнуть — сейчас все увидите на примере.

Для пояснения перечислений лучшим примером могут служить дни недели. Допустим, нам нужно иметь переменную, в которой надо сохранить текущий день недели. Как это можно сделать? Сначала следует просто понять, что такое день недели и в каком типе данных его представлять. Можно представить его строкой, но это будет уже не день недели, а всего лишь *название* дня недели. Можно представить его числом от 1 до 7 или от 0 до 6 (кому как удобнее), но это будет *номер* дня, но не день недели. Как же тогда быть? Почему разработчики Visual Studio не позаботились о такой ситуации и не внедрили тип данных, который являлся бы именно днем недели? Возможно, и внедрили, но это не важно, потому что подобные типы данных мы можем легко создавать сами с помощью перечислений `enum`.

Итак, объявление перечисления `enum` выглядит следующим образом:

```
enum имя { Значения через запятую };
```

Наша задача по созданию типа для хранения дня недели сводится к следующей строке кода:

```
enum WeekDays { Monday, Tuesday, Wednesday,  
    Thursday, Friday, Saturday, Sunday };
```

Вот и все. В фигурных скобках записаны имена дней недели на английском — вполне понятные имена, которые можно использовать в приложении. Теперь у нас есть новый тип данных `WeekDays`, так что мы можем объявлять переменные этого типа и присваивать им значения дней недели. Например:

```
WeekDays day;  
day = WeekDays.Thursday;
```

В первой строке мы объявили переменную `day` типа `WeekDays`. Это объявление идентично созданию переменных любого другого типа. Во второй строке переменной присваивается значение четверга. Как это делается? Нужно просто написать тип данных, а через точку указать то значение перечисления, которое вы хотите присвоить: `WeekDays.Thursday`.

Чтобы показать перечисления во всей красе, я написал небольшой пример, который иллюстрирует различные варианты их использования. Код примера приведен в листинге 2.1.

Листинг 2.1. Пример работы с `enum`

```
// массив с названиями дней недели на русском
string[] WeekDaysRussianNames = { "Понедельник", "Вторник",
    "Среда", "Четверг", "Пятница", "Суббота", "Воскресенье" };

WeekDays day = WeekDays.Thursday;

// вывод дня недели в разных форматах
Console.WriteLine("Сегодня " + day);
Console.WriteLine("Сегодня " + WeekDaysRussianNames[(int)day]);
int dayIndex = (int)day + 1;
Console.WriteLine("Номер дня " + dayIndex);

// вот так можно делать проверку сравнением
if (day == WeekDays.Friday)
    Console.WriteLine("Скоро выходной");
Console.ReadLine();

// объявление enum
enum WeekDays { Monday, Tuesday, Wednesday,
    Thursday, Friday, Saturday, Sunday };
```

Сразу стоит дать очень важный комментарий по поводу организации этого кода. Объявление `enum` может находиться в самом конце файла или в самом начале. Если вы работаете с операторами верхнего уровня, то в файле `Program.cs` все объявления нужно делать только в самом конце. Если сделать это вначале, то произойдет ошибка. Поэтому объявление `enum` находится в конце кода.

Чтобы было проще понять код примера, я добавил в него подробные комментарии. Сразу скажу, что если запустить это приложение, то на экране будет отображено:

```
Сегодня Thursday
Сегодня Четверг
Номер дня 4
```

Значения перечислений могут быть написаны только на английском и не могут содержать пробелы, поэтому при выводе на экран их имена могут показаться пользователю несколько недружественными. Чтобы сделать их дружественными, в этом

примере я создал массив из названий дней недели на русском и использовал его для превращения типа данных `WeekDays` в дружественное пользователю название.

После этого идет создание переменной типа `WeekDays`, и тут же ей присваивается значение четверга. Не знаю, почему я выбрал этот день, ведь сегодня на календаре среда, но что-то мне захотелось поступить именно так.

Далее идет самое интересное — вывод значения переменной `day`. В первой строке вывода я просто отправляю ее как есть на консоль. По умолчанию переменная будет превращена в строку (магия метода `ToString()`, о котором мы будем говорить в разд. 3.10), поэтому четверг на экране превратится в `Thursday`.

Следующей строкой я хочу вывести на экран название дня недели на русском. Для этого из массива `WeekDaysRussianNames` нужно взять соответствующую дню недели строку. Задачу упрощает то, что в массиве и в перечислении `enum` индексы значений каждого дня недели совпадают: в перечислении четверг имеет индекс 3 (перечисления, как и массивы, нумеруются с нуля), и в массиве названия дней недели нумеруются с нуля. Теперь нам нужно просто узнать индекс текущего значения переменной `day`, а для этого достаточно перед переменной поставить в круглых скобках тип данных `int` (этот метод называется *приведением типов*, о чем мы еще поговорим отдельно в разд. 6.1) вот так: `(int)day`. Этим мы говорим, что нам нужно не название дня, а именно индекс. Полученный индекс указываем в качестве индекса массива и получаем имя дня недели на русском:

```
WeekDaysRussianNames[(int)day]
```

Теперь я хочу отобразить на экране номер дня недели. Мы уже знаем, что для получения индекса нужно перед именем поставить в скобках тип данных: `int`. Но индекс нумеруется с нуля, поэтому я прибавляю единицу, чтобы четверг был четвертым днем недели, как привыкли люди в жизни, а не компьютеры в виртуальности.

Напоследок я показываю, как переменные типа перечислений `enum` можно использовать в операторах сравнения. Даже не знаю, что тут еще добавить... Мне кажется, что код намного красноречивее меня.

По умолчанию элементы перечисления получают индексы (значения) от 0. А что если мы хотим, чтобы они имели индексы, начиная со ста? В этом случае можно первому элементу присвоить нужный нам индекс. Это делается простым присваиванием:

```
enum MyColors
{
    Red = 100,
    Green,
    Blue
}
```

Я тут завел новое перечисление цветов, чтобы оно было поменьше по размеру. Теперь, если привести красный цвет к числу, мы увидим 100, в случае с зеленым — увидим 101, а для синего это будет 102.

Вы можете назначить каждому элементу свои собственные индексы:

```
enum MyColors
{
    Red = 100,
    Green = 110,
    Blue = 120
}
```

2.5. Простейшая математика

Переменные заводятся для того чтобы производить с ними определенные вычисления. Мы пока не будем углубляться в сложные математические формулы, но на простейшую математику посмотрим.

В языке C# есть следующие математические операторы:

- ☐ сложение: +;
- ☐ вычитание: -;
- ☐ умножение: *;
- ☐ деление: /.

Давайте заведем переменную `i`, которой присвоим сначала значение 10, а потом умножим эту переменную на 2 и разделим на 5. В коде это будет выглядеть следующим образом:

```
int i;
i = 10;
i = i * 2 / 5;
Console.WriteLine(i);
```

Как видите, писать математические вычисления не так уж и сложно — практически так же, как и в школе в тетради. Переменной `i` мы просто с помощью знака равенства присваиваем результат математических вычислений.

В программировании можно взять переменную, изменить ее и записать результат обратно в эту же переменную.

Существуют и сокращенные варианты математических операций, позаимствованные из языка C++:

- ☐ увеличить значение переменной на 1: ++;
- ☐ уменьшить значение переменной на 1: --;
- ☐ прибавить к переменной: +=;
- ☐ вычесть из переменной: -=;
- ☐ умножить переменную на значение: *=;
- ☐ разделить переменную на значение: /=.

Вот тут нужны некоторые комментарии. Допустим, мы хотим увеличить значение переменной `i` на 1. Для этого можно написать следующие строки кода:

```
int i = 10;
i = i + 1;
```

В первой строке кода мы объявляем переменную `i` и присваиваем ей значение 10. Во второй строке значение переменной увеличивается на 1, и мы получаем в результате число 11. А можно использовать более короткий вариант:

```
int i = 10;
i++;
```

В первой строке кода также объявляется переменная `i`, а во второй строке значение этой переменной увеличивается на 1 с помощью оператора `++`.

Теперь посмотрим, как можно уменьшать значение переменной с помощью оператора `--`:

```
int i = 10;
i--;
```

В результате в переменной `i` после выполнения этого кода будет находиться число 9. Это то же самое, что написать: `i = i - 1`.

Если нужно увеличить значение переменной на значение, отличное от 1, то можно воспользоваться оператором `+=`. Записывается он следующим образом:

```
переменная += значение;
```

Это означает, что к переменной нужно прибавить значение, а результат записать обратно в переменную.

Например:

```
int i = 10;
i += 5;
```

Здесь объявляется переменная `i` и тут же ей присваивается значение 10. Во второй строке значение переменной увеличивается на 5.

Точно так же можно воспользоваться и умножением. Если нужно умножить переменную `i` на значение — например, на 5, то пишем следующую строку:

```
i *= 5;
```

Разумеется, можно уменьшать переменные на определенные значения или делить.

Если результат сложения чисел (целых и вещественных) предсказуем, то возникает очень интересный вопрос: а что произойдет, если сложить две строки? Математически сложить строки невозможно, так, может быть, эта операция невыполнима? Да нет, сложение строк вполне даже возможно, просто вместо математического сложения происходит *конкатенация*. Конкатенация — это когда одна строка добавляется (не прибавляется математически, а добавляется/присоединяется) в конец

другой строки. Так, следующий пример складывает три строки для получения одного текстового сообщения:

```
string str1 = "Hello,";  
string str2 = "World";  
string strresult = str1 + " " + str2;  
Console.WriteLine(strresult);  
Console.ReadLine();
```

В результате на экране появится сообщение: **Hello, World.**

А вот операции вычитания, умножения и деления со строками невозможны, потому что таких математических операций не существует, а нематематической операции со схожим смыслом я даже представить себе не могу. Нет, у меня, конечно же, не математическое образование, и, возможно, что-то подобное в сфере высокой математики существует, но умножения и деления строк в C# пока все же нет.

Помните, мы говорили в *разд. 2.4.1*, что строки никогда не изменяются, а всегда создаются заново. Посмотрите на следующий код:

```
string str1 = "Hello,";  
str1 = str1 + " World";
```

В первой строке мы объявляем переменную и присваиваем ей значение. Во второй строке переменной `str1` присваиваем результат сложения этой самой переменной `str1` и еще одного слова. Да, эта операция вполне законна, и в таком случае система сложит строки, подсчитает память, необходимую для хранения результата, и заново проинициализирует `str1` с достаточным объемом памяти для хранения результата конкатенации.

Математика в C# и, вообще, в программировании не так уж и сложна и идентична всему тому, что мы изучали в школе. Это значит, что классическая задача $2 + 2 * 2$ решается компьютером так же, как и человеком, — результат будет равен 6 или около того. Правда, у некоторых людей иногда может получиться и 8, но это случается не так уж и часто. Приоритетность выполнения операций у компьютера та же, что мы изучали на уроках математики, а это значит, что он сначала выполнит умножение и только потом сложение, — вне зависимости от того, как вы это записали: $2 * 2 + 2$ или $2 + 2 * 2$.

Если необходимо сначала выполнить сложение, и только потом умножение, то нужно использовать круглые скобки, которые имеют более высокий приоритет, и написать код так:

```
int result = (2 + 2) * 2;
```

Вот теперь результатом будет 8.

Я заметил, что с этим сложностей у пользователей не бывает, а основную проблему вызывает сокращенный *инкремент* или *декремент*, т. е. операции соответственно `++` и `--`. Если вы просто написали `i++`, то никаких вопросов и проблем нет: переменная `i` просто увеличивается на 1. Проблемы возникают, когда оператор `++` или `--` пытается использовать в выражениях. Посмотрите на следующий код:

```
int i = 1;  
int j = i++;
```

Как вы думаете, чему будут равны переменные `i` и `j` после его выполнения? Те, кто не имел опыта работы с инкрементом, считают, что переменная `i` будет равна 1, а `j` станет равна 2. Наверное, это связано с тем, что людям кажется, будто оператор `++` увеличивает переменную и как бы возвращает ее результат. Но инкремент ничего не возвращает, поэтому результат получается ровно обратный: `i` будет равно 2, а `j` будет равно 1.

Запомните, что если плюсы или минусы стоят *после* переменной, то во время вычисления будет использоваться текущее значение переменной `i` (а значит, в `j` будет записана 1), а увеличению окажется подвержена именно переменная `i` после выполнения расчетов в операторе (т. е. только `i` будет увеличена на 1).

Если вы хотите, чтобы переменная `i` увеличилась до выполнения расчета, то нужно поставить плюсы *перед* переменной `i`:

```
int j = ++i;
```

Выполняя этот код, программа сначала увеличит переменную `i` до 2, а потом присвоит это значение `i` (уже увеличенное) переменной `j`, а это значит, что обе переменные будут равны 2. Получается, что в обоих случаях в выражении просто используется текущее значение переменной: если `++` стоит *перед* переменной, то она увеличивается *до расчетов*, а если *после*, то увеличение произойдет *после расчетов*. А если вам нужно присвоить переменной `j` значение, на 1 большее, чем `i`, и при этом не надо увеличивать саму переменную `i`, то писать `i++` или `++i` нельзя, — следует использовать классическое математическое сложение с 1:

```
int j = i + 1;
```

Попробуйте теперь сказать, что будет выведено на экран после выполнения следующего кода:

```
int i = 1;  
Console.WriteLine("i = " + i++);
```

В результате в консоли будет выведено: `i = 1`, а переменная `i` станет равна 2. Дело в том, что в консоль пойдет текущее значение переменной, а только после этого `i` будет увеличена до 2.

А теперь еще вопрос на засыпку: что будет, если выполнить следующую строку кода:

```
i = i++;
```

Вообще ничего не изменится. Ответ на первый взгляд немного нелогичен, но, с другой стороны, здесь ничего из того, что мы только что узнали, не нарушается. Переменной `i` присваивается текущее значение, и именно оно является результатом, а то, что после этого инкремент `++` увеличивает число на 1, ни на что не влияет. Дело в том, что переменная `i` уже рассчитана при выполнении присваивания, а

операция инкремента переменной `i` будет просто потеряна (перезаписана результатом выполнения присвоения).

Надо быть очень внимательным, используя операторы `++` и `--` в проектах, потому что их неправильное применение приведет к неверной работе программы или даже к ошибке, и иногда такие ошибки увидеть с первого взгляда непросто.

Необходимо заметить еще одну очень важную особенность математики в C# — размерность расчетов. При вычислении операторов C# выбирает максимальную размерность используемых составляющих. Например, если происходит вычисление с участием целого числа и вещественного, то результатом будет вещественное число. Это значит, что следующий код не будет откомпилирован:

```
int d = 10 * 10.0;
```

Несмотря на то что дробная часть второго числа равна нулю и результат должен быть 100, что вполне приемлемо для целочисленного типа `int`, компилятор выдаст ошибку. Если хотя бы один из операндов имеет тип `double`, результат нужно записывать в переменную `double` или использовать приведение типов, о чем мы пока ничего не знаем, но все еще впереди.

Если тип данных всех составляющих одинаковый, то выбирается максимальная размерность. Например:

```
int i = 10;  
byte b = 10 * i;
```

Во второй строке мы пытаемся сохранить результат перемножения числа 10 и переменной типа `int` в переменной типа `byte`. Эта переменная может принимать значения от 0 до 255, и, по идее, результат перемножения 10×10 должен в нее поместиться, но компилятор не будет этого выяснять. Он видит, что одна из составляющих равна `int`, и требует, чтобы результат тоже записывался в максимальный тип `int`.

Усложняем задачу и смотрим на следующий пример:

```
long l = 1000000 * 1000000;
```

В этом примере перемножаются миллион и миллион. Результат будет слишком большим для числа `int`, поэтому в качестве переменной результата был выбран тип `long`. Вроде бы все корректно, но в результате компиляции опять будет выдана ошибка. В этом случае компилятор размышляет следующим образом: он видит числа, а все числа по умолчанию он воспринимает как `int`. Когда оба числа `int`, он рассчитывает результат именно в `int` и только потом записывает результат в переменную `long`. А поскольку компилятор не может рассчитать значение перемножения миллиона на миллион в памяти как `int`, то происходит ошибка.

Необходимо, чтобы в расчете участвовала хотя бы одна переменная типа `long`, тогда система будет и в памяти рассчитывать результат как `long`. Для этого можно ввести в пример дополнительную переменную, а можно просто после числа указать букву `L`, которая как раз и укажет компилятору, что перед ним находится длинное целое число:

```
long l = 1000000 * 1000000L;
```

Вот такой пример откомпилируется без проблем, потому что справа от знака умножения находится число, которое компилятор явно воспринимает как длинное целое `long` и выберет этот тип данных как максимальный.

2.6. Условные операторы

Очень часто в программах необходима определенная логика, с помощью которой можно повлиять на процесс выполнения программы. Под *логикой* тут понимается выполнение определенных операций в зависимости от каких-либо условий.

Условные операции строятся вокруг типа данных `bool`. Этот тип может принимать всего два значения: `true` или `false` (по-русски: истина или ложь). Следующая строка кода объявляет переменную и присваивает ей истинное значение:

```
bool variable = true;
```

В переменную можно сохранять результат сравнения. Ведь что такое сравнение, например, на равенство? Если два значения одинаковые, то результатом сравнения будет истина (результат сравнения верный), иначе — ложь (результат неверный).

2.6.1. Условный оператор *if*

Для создания логики приложения в C# используется оператор `if`, синтаксис которого в общем виде таков:

```
if (Условие)
    Действие 1;
else
    Действие 2;
```

После оператора `if` в скобках пишется условие. Если условие верно (истинно), то выполнится действие 1, иначе будет выполнено действие 2.

Второе действие является необязательным. Вы можете ограничиться только проверкой на верность условия, и в этом случае оператор будет выглядеть так:

```
if (Условие)
    Действие 1;
```

Обращаю ваше внимание, что в одном операторе `if` будет выполняться только одно действие. Если необходимо выполнить несколько операторов, то их нужно заключить в фигурные скобки:

```
if (Условие)
{
    Действие 1;
    Действие 2;
    ...
}
```

Если в этом примере забыть указать фигурные скобки:

```
if (Условие)
    Действие 1;
    Действие 2;
...
```

то при верности условия будет выполнено только *Действие 1*, а остальные действия будут выполнены в любом случае, даже при ложном значении условия. Таким образом, как бы красиво ни был отформатирован код и задано условие, без фигурных скобок выполнятся все записанные действия. А фигурные скобки как бы объединяют несколько действий в одно, т. е. представляют собой одно действие, хотя внутри этих скобок может быть сколько угодно команд:

```
if (Условие)
{
}
}
```

В качестве условия необходимо указать один из операторов сравнения. В C# поддерживаются следующие операторы:

- ☐ больше: >;
- ☐ меньше: <;
- ☐ больше либо равно: >=;
- ☐ меньше либо равно: <=;
- ☐ равно: ==;
- ☐ не равно: !=.

Обратите внимание, что оператор сравнения на равенство — это *два* символа равно. Один такой символ — это еще присваивание, а два — это уже сравнение.

Допустим, что необходимо уменьшить значение переменной на единицу только в том случае, если она больше 10. Такой код будет выглядеть следующим образом:

```
int i = 14;
if ( i > 10 )
    i--;
```

Сначала мы заводим переменную *i* и присваиваем ей значение 14. Затем проверяем, и если переменная больше 10, то уменьшаем ее значение на единицу с помощью оператора *i--*.

Усложним задачу: если переменная больше 10, то уменьшаем ее значение на 1, иначе увеличиваем значение на 1:

```
int i = 14;
if ( i > 10 )
    i--;
else
    i++;
```

В обоих случаях выполняется только одно действие. Если действий должно быть несколько, то объединяем их в фигурные скобки:

```
int i = 14;
if ( i > 10 )
{
    i--;
    Console.WriteLine(i);
}
else
    i--;
```

Здесь если переменная больше 10, то выполняются два действия: значение переменной уменьшается и тут же выводится на экран консоли. Иначе значение переменной только уменьшается.

А что если нужно выполнить действие, когда условие, наоборот, не выполнено? Допустим, что у нас есть проверка ($i > 0$), но мы хотим выполнить действие, когда это условие ложное. В таком случае можно развернуть условие следующим образом ($i \leq 0$) или инвертировать его с помощью символа восклицательного знака: $!(i \leq 0)$. Этот символ меняет булево значение на противоположное, т. е. `true` на `false` и наоборот.

```
int i = 0;
if (!(i > 0))
    Console.WriteLine("Переменная i равна или меньше нуля");
```

А что если нужно сделать сразу две проверки? Для этого используются *логические операторы*:

□ И: `&&`;

□ ИЛИ: `||`;

□ НЕ: `!`.

Как они работают? Допустим, нам нужно убедиться, что число `i` больше 0 и меньше 10. Очень популярная задача, особенно с массивами, потому что если у вас есть массив из 10 элементов, а вы попытаетесь обратиться к 11-му, то в программе произойдет ошибка. Очень важно проверить, что индекс, который мы хотим использовать, находится в допустимых пределах, и такую двойную проверку за один шаг можно сделать с помощью оператора `&&` (И):

```
int[] a = new int[10];
int i = 20;
if (i >= 0 && i < 10)
{
    // все в порядке, индекс верный
    int value = a[i];
}
```

В этом примере мы проверяем с помощью `if` переменную `i`, чтобы значение было больше или равно 0 И меньше 10. Именно оба условия должны быть одновременно выполнены.

Оператор `||` поможет проверить логическое ИЛИ. Давайте с его помощью решим ту же самую задачу:

```
int[] a = new int[10];
int i = 20;
if (i < 0 || i >= 10)
{
    // Ошибка, индекс не верный
}
```

Здесь мы ищем обратный результат — неверный индекс. Если `i` меньше нуля ИЛИ если `i` больше 10, то индекс для нас не корректный, и в этом случае будет выполнен следующий оператор.

И последний оператор — `!` (НЕ). Он меняет логику на обратную. Например, следующая строка проверяет, равно ли число `i` десяти:

```
if (i == 10)
```

А если мы хотим написать не равно, то можно использовать `!=` или просто символ `!` следующим образом:

```
if (!(i == 10))
```

Выглядит страшновато, и, наверное, поэтому никто не использует такой подход. Вместо него намного более читаемым вариантом выглядит:

```
if (i != 10)
```

2.6.2. Условный оператор *switch*

Когда нужно выполнить несколько сравнений подряд, мы можем написать что-то в стиле:

```
if (i == 1)
    Действие 1;
else if (i == 2)
    Действие 2;
else if (i == 3)
    Действие 3;
```

Никто не запрещает нам делать так, но не кажется ли вам, что этот код немного страшноват? На мой взгляд, в нем присутствует определенное уродство, поэтому в таких случаях я предпочитаю использовать другой условный оператор — `switch`, синтаксис которого в общем виде таков:

```
switch (переменная)
{
```

```
case Значение1:
    Действия (может быть много);
    break; // указывает на конец ветки
case Значение2:
    Действия (может быть много);
    break; // указывает на конец ветки
default:
    Действия по умолчанию;
    break; // указывает на конец ветки
}
```

Этот код выглядит немного приятнее, если к нему привыкнуть. Программа последовательно сравнивает значение переменной со значениями и, если находит совпадение, выполняет соответствующие действия. Если ничего не найдено, то будет выполнен код, который идет после ключевого слова `default`. Действие по умолчанию не является обязательным, и этот фрагмент кода можно опустить.

Следующий пример показывает, как можно проверить числовую переменную на возможные значения от 1 до 3. Если ничего не найдено, то будет выведено текстовое сообщение:

```
switch (i)
{
    case 1:
        Console.WriteLine("i = 1");
        break;
    case 2:
        Console.WriteLine("i = 2");
        break;
    case 3:
        Console.WriteLine("i = 3");
        break;
    default:
        Console.WriteLine("Ну не понятно же!");
        break;
}
```

Оператор `case` очень удобен, когда нужно сравнить переменную на несколько возможных значений. Впрочем, если вы предпочитаете использовать несколько операторов `if` подряд, то в этом нет ничего плохого.

2.6.3. Сокращенная проверка

Язык C# взял все лучшее из C++ и получил очень хороший метод короткой проверки логической операции:

Условие ? Действие 1 : Действие 2

Без каких-либо прелюдий мы сразу же пишем условие, которое должно проверяться. Затем, после символа вопроса, пишем действие, которое должно быть выполнено в случае истинного результата проверки, а после двоеточия пишем действие, которое должно быть выполнено при неудачной проверке.

Такой метод очень удобен, когда вам нужно произвести проверку прямо внутри какого-то кода. Например, в следующем коде проверка происходит прямо внутри скобок, где мы всегда указывали, что должно выводиться на экран:

```
int i = 10;
Console.WriteLine(i == 10 ? "i = 10" : "i != 10");
Console.WriteLine(i == 20 ? "i = 20" : "i != 20");
```

В обоих случаях переменная `i` проверяется на разные значения, и, в зависимости от результата проверки, будет выведено в консоль или сообщение после символа вопроса (в случае удачной проверки), или после символа двоеточия (если проверка неудачная).

2.7. Циклы

Допустим, нам нужно несколько раз выполнить одну и ту же операцию. Для этого можно столько раз, сколько надо, подряд написать один и тот же код, и никаких проблем. А если операция должна выполняться 100 раз? Вот тут возникает проблема, потому что писать 100 строчек кода нудно, неинтересно и абсолютно неэффективно. А если придется изменить формулу, которая выполняется 100 раз? После такого не захочется больше никогда программировать.

Проблему решают *циклы*, которые выполняют указанное действие определенное количество раз.

2.7.1. Цикл *for*

В общем виде синтаксис цикла `for` таков:

```
for (Инициализация; Условие; Порядок выполнения)
    Действие;
```

После оператора `for` в скобках указываются три оператора, разделенные точкой с запятой:

- ❑ *Инициализация* — начальное значение переменной счетчика;
- ❑ *Условие* — пока это условие возвращает истину, действие будет выполняться;
- ❑ *Порядок выполнения* — команда, которая должна увеличивать счетчик.

Итак, пока условие, указанное в скобках посередине, верно, будет выполняться действие. Обратите внимание, что циклически будет выполняться только одна команда. Если необходимо выполнить несколько действий, то их нужно заключить в фигурные скобки — точно так же, как мы это делали с логическими операциями:

```
for (Инициализация; Условие; Порядок выполнения)
{
    Действие 1;
    Действие 2;
}
```

Действия, которые выполняет цикл, еще называют *телом цикла*.

Пора обратиться к примеру. Давайте посмотрим, как можно рассчитать факториал числа 5. В учебных целях факториалы очень удобны, поэтому я всегда рассматриваю их при описании циклов.

Что такое *факториал*? Это результат перемножения чисел от 1 до указанного числа. Факториал числа 5 — это результат перемножения $1 * 2 * 3 * 4 * 5$. Можно явно прописать эту формулу, но это слишком просто и неуниверсально. Более эффективным решением будет использование цикла. Итак, следующий пример показывает, как можно рассчитать факториал числа 5:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; i++)
{
    sum = sum * i;
}
Console.WriteLine(sum);
```

До начала цикла объявляются две целочисленные переменные: `sum` и `max`. Первая из этих переменных используется при расчете факториала, а вторая — определяет максимальное значение, до которого нужно перебирать математический ряд.

Переходим к рассмотрению цикла. В скобках оператора `for` в первом операторе объявлена переменная `i`, которой присваивается значение 2. Действительно, в операторе цикла тоже можно объявлять переменные, но тут нужно забежать вперед и сказать про *область видимости* такой переменной, — она будет доступна только внутри цикла. За пределами цикла переменная не будет видна. Например, следующий код будет ошибочным:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; i++)
{
    sum *= i;
}
Console.WriteLine(i);
```

Здесь после цикла вызывается метод `WriteLine()`, который пытается вывести в консоль значение переменной `i`. Если попытаться скомпилировать этот проект, то компилятор выдаст ошибку и сообщит нам, что переменной `i` не существует. Если необходимо видеть переменную и за пределами цикла, то ее нужно объявить перед циклом:

```
int sum = 1, max = 5, i;
for (i = 2; i <= max; i++)
```



```
{  
    sum *= i;  
}  
Console.WriteLine(i);
```

Вот теперь переменная `i` объявлена до цикла, а в скобках оператора `for` только задается ее значение. Теперь значение переменной окажется доступно и за пределами цикла, и код будет корректным.

Почему цикл расчета факториала начинается с 2, а не с нуля или с единицы? Интересный вопрос, но на него я отвечу чуть позже.

Второй оператор цикла `for` — условие (`i <= max`). Это значит, что цикл будет выполняться от 2 (это мы задали в первом параметре) и до тех пор, пока значение переменной `i` не станет больше значения переменной `max`. В условии не обязательно использовать переменную, можно было просто написать `i <= 5`.

```
int sum = 1;  
for (int i = 2; i <= 5; i++)  
{  
    sum = sum * i;  
}  
Console.WriteLine(sum);
```

Последний оператор цикла определяет, как будет изменяться счетчик. В нашем случае переменная `i` увеличивается на единицу — т. е. на каждом этапе к счетчику будет прибавляться единица. На самом деле тут может быть любое математическое выражение, например `i + 2`, если мы хотим, чтобы значение увеличивалось на 2.

Теперь посмотрим логику выполнения цикла. Когда начинает выполняться цикл, то переменная `i` изначально равна 2, а переменная `sum` равна 1. Это значит, что после выполнения действия: `sum *= i` (это то же самое, что написать `sum = sum * i`) в переменную `sum` будет записан результат перемножения 1 и 2.

Потом программа увеличит значение счетчика `i` на единицу (т. е. выполнит операцию `i++`). После увеличения счетчика происходит проверка, и если счетчик превысил или стал больше значения `max`, то цикл прерывается и выполнение программы продолжается за пределами цикла. В нашем случае счетчик на втором шаге становится равным 3, а значит, нужно продолжать выполнение цикла. Снова выполняется действие `sum *= i`. После выполнения первого шага цикла переменная `sum` равна 2, и, следовательно, произойдет умножение этого числа на значение счетчика, т. е. на 3. Результат (6) будет записан в переменную `sum`.

Снова увеличиваем счетчик на 1 и производим проверку. И снова счетчик еще не превысил или не стал равен значению `max`. Что произойдет дальше, уже не сложно догадаться.

Таким образом, тело цикла выполняется от 2 до 5, т. е. 4 раза со значениями счетчика 2, 3, 4, 5. Как только счетчик станет равным 6, он превысит значение переменной `max`, и его выполнение прервется.

Значение счетчика можно увеличивать и в теле цикла. Например:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; )
{
    sum *= i;
    i++;
}
Console.WriteLine(sum);
```

В скобках после `for` третий параметр пуст, и счетчик не увеличивается циклом, зато он изменяется в теле цикла. Вторая команда тела цикла как раз и увеличивает счетчик.

2.7.2. Цикл *while*

Цикл `while` выполняется, пока условие верно. В общем виде синтаксис `while` таков:

```
while (условие)
    Действие;
```

Цикл выполняет только одно действие. Если необходимо выполнить несколько действий, то их нужно объединить фигурными скобками:

```
while (условие)
{
    Действие 1;
    Действие 2;
    ...
}
```

Посмотрим на пример расчета факториала с помощью цикла `while`:

```
int sum = 1, max = 5;
int i = 2;
while (i <= max)
{
    sum *= i;
    i++;
}
Console.WriteLine(sum);
```

Надеюсь, что этот пример будет красноречивее любых моих слов. Так как цикл может только проверять значение переменной, то начальное значение мы должны задать до начала цикла, а увеличивать счетчик нужно в теле цикла. Не забывайте про увеличение. Посмотрите на следующий код и попробуйте сразу на глаз определить ошибку, и к чему она приведет:

```
int sum=1, max=5;
int i = 2;
```

```
{  
    while (i<=max)  
        sum *=i;  
        i++;  
}
```

Я поставил здесь фигурные скобки так, что они не несут теперь никакой смысловой нагрузки и ни на что не влияют. После цикла нет фигурных скобок, а значит, будет выполняться только одно действие — увеличение `sum` в `i` раз. Увеличения переменной `i` происходить не будет, т. е. она никогда не превысит число 5, и цикл никогда не завершится — он станет бесконечным. Вывод? Надо быть внимательным при написании циклов и обязательно следует убедиться, что когда-нибудь наступит ситуация, при которой цикл прервется.

2.7.3. Цикл *do...while*

Цикл `while` имеет одно ограничение — если условие заведомо неверно, то действие не будет выполнено вообще ни разу. Иногда бывает необходимо выполнить действие один раз — вне зависимости от результата проверки условия. В этом случае можно воспользоваться циклом `do...while`, который выглядит так:

```
do  
    Действие;  
while (условие);
```

Я думаю, уже не нужно пояснять, что выполняется только одно действие, и что необходимо сделать, если надо выполнить в цикле несколько операций.

Для начала заметим одно важное различие — после скобок оператора `while` стоит точка с запятой. Второе различие — условие стоит после действия. Это значит, что сначала выполняется действие, а потом уже проверяется условие, — следующий шаг цикла будет выполнен, только если условие выполнено.

Посмотрим, как выглядит расчет факториала с помощью оператора `do...while`:

```
int sum = 1, max = 5;  
int i = 2;  
do  
{  
    sum *= i;  
    i++;  
} while (i <= max);  
Console.WriteLine(sum);
```

А что если нужно с помощью этого кода вычислить факториал числа 1? Факториал единицы равен единице, но если мы просто изменим `max` на 1, код вернет 2, ведь первый шаг цикла выполняется вне зависимости от проверки, а значит, цикл успеет умножить переменную на 2. Поэтому цикл `do...while` лучше не использовать для вычисления факториала, иначе он выдаст неверный результат для значения 1.

2.7.4. Цикл *foreach*

С помощью циклов очень удобно обрабатывать массивы значений. Допустим, у нас есть массив целых чисел, и нам нужно найти в нем максимальное и минимальное значения. Для начала посмотрим, как можно решить эту задачу с помощью цикла `for`:

```
int[] array = { 10, 20, 4, 19, 44, 95, 74, 28, 84, 79 };

int max = array[0];
int min = array[0];

for (int i = 0; i < 10; i++)
{
    if (array[i] < min)
        min = array[i];
    if (array[i] > max)
        max = array[i];
}

Console.WriteLine("Максимальное значение " + max);
Console.WriteLine("Минимальное значение " + min);
```

Для начала мы здесь объявляем и тут же инициализируем массив `array` десятью значениями. После этого объявляются две целочисленные переменные: `max` и `min`, которым по умолчанию присваивается значение нулевого элемента из массива. Я знаю, что массивы не пустые, поэтому могу так поступить. Если вы получаете данные от пользователя, то желательно проверять, чтобы массивы не были пустыми. Если там вообще нет элементов, то обращение к нулевому элементу пустого массива во время выполнения программы приведет к ошибке.

Теперь запускаем цикл, который будет выполняться от 0 и пока `i` меньше 10, т. е. максимум до 9, — именно такие значения может принимать индекс элементов в нашем массиве.

Внутри цикла сначала проверяем, меньше ли текущий элемент (`array[i]`) минимального, и, если это так, сохраняем текущее значение в переменной `min`. После этого такую же проверку делаем на максимальное значение. Обратите внимание, что после оператора `if` нет фигурных скобок, и это логично, потому что надо выполнить только одно действие.

Когда нужно работать со всем содержимым массива, я предпочитаю использовать цикл `foreach`. В общем виде синтаксис этого цикла таков:

```
foreach (тип переменная in массив)
    Действие;
```

В круглых скобках сначала мы описываем тип и переменную, через которую на каждом этапе цикла будем получать доступ к очередному значению, и указываем массив, все значения которого хотим просмотреть. Тип данных для переменной

должен быть точно таким же, каким являются элементы массива. Если перед нами массив целых чисел, то и переменная должна иметь тип целого числа.

Теперь посмотрим, как будет выглядеть цикл `foreach` для поиска максимального и минимального элементов в массиве числовых значений:

```
int[] array = { 10, 20, 4, 19, 44, 95, 74, 28, 84, 79 };

int max = array[0];
int min = array[0];

foreach (int value in array)
{
    if (value < min)
        min = value;
    if (value > max)
        max = value;
}
```

Здесь мы также объявляем массив и инициализируем начальные значения для переменных результата. Самое интересное происходит в скобках после `foreach`, где описывается переменная с именем `value` (имя, конечно же, может быть любым) типа `int`, потому что массив у нас из целых чисел.

Внутри цикла мы обращаемся к текущему элементу массива через переменную `value`, именно ее значение сравниваем с максимальным и минимальным значениями, и при необходимости сохраняем это значение в максимальном или в минимальном значении.

Преимущество цикла `foreach` в том, что вы не выйдете за пределы массива и вам не нужно задумываться о том, сколько элементов находится в массиве. Вы всегда просмотрите все элементы.

2.8. Управление циклом

Циклы — достаточно мощное и удобное решение для множества задач, но они еще мощнее, чем вы думаете, потому что ходом выполнения цикла можно управлять. Давайте познакомимся с операторами, которые позволяют управлять ходом выполнения цикла.

2.8.1. Оператор *break*

Первый оператор — `break`. Как только программа встречает этот оператор, она прерывает цикл:

```
int sum = 1, max = 5;
for (int i = 2; ; )
{
    sum *= i;
    i++;
}
```

```
    if (i > max)
        break;
}
```

После оператора `for` второй оператор в скобках пустой. Это значит, что проверки не будет, и такой цикл будет выполняться вечно. Если нет проверки, то нет и возможности прервать цикл. Но если посмотреть на тело цикла, то вы увидите, что там происходит проверка. Если переменная `i` больше значения `max`, то выполняется оператор `break`, т. е. работа цикла прерывается. Это значит, что цикл будет проходить значения счетчика от 2 до 5 включительно.

2.8.2. Оператор *continue*

Еще один оператор, который позволяет управлять циклом, — `continue`. Этот оператор прерывает текущий шаг цикла и заставляет перейти на выполнение следующего шага. Например, вы хотите перемножить числа от 1 до 5, пропустив при этом число 4. Это можно выполнить следующим циклом:

```
int sum = 1, max = 5;
for (int i = 2; ; )
{
    if (i == 4)
    {
        i++;
        continue;
    }

    sum *= i;
    i++;

    if (i > max)
        break;
}
```

В этом примере перед тем, как произвести перемножение, происходит проверка. Если текущее значение счетчика равно 4, то тело цикла дальше выполняться не будет, а произойдет увеличение счетчика и переход на начало выполнения следующего шага. При этом, если до оператора `continue` есть какие-либо действия, они будут выполнены. Например:

```
int sum = 1, max = 5;
for (int i = 2; i <= max; )
{
    sum *= i;
    i++;

    if (i == 4)
        continue;
}
```

Здесь сначала переменная `sum` умножается на счетчик, и только потом произойдет проверка на равенство счетчика числу 4. В этом случае очень важно, что счетчик увеличивается до проверки. Дело в том, что он не увеличивается автоматически (третий оператор в скобках после `for` пуст), и следующий цикл будет бесконечным:

```
int sum=1, max=5;
for (int i = 2; i<=max ;)
{
    sum *=i;

    if (i == 4)
        continue;

    i++;
}
```

Если счетчик `i` равен 4, то дальнейшего выполнения тела цикла не будет. При переходе на следующий шаг счетчик также не будет увеличен, а значит, опять `i` будет равен 4 и снова выполнится оператор `continue`. Так будет продолжаться бесконечно, потому что `i` не сможет увеличиваться и превысить значение переменной `max`.

2.9. Константы

Константы — это такие переменные, значения которых нельзя изменить во время выполнения программы. Значение задается только во время их объявления и после этого не изменяется. Чтобы переменная стала константой, в объявление нужно добавить ключевое слово `const`:

```
const double Pi = 3.14;
double Pi2 = Pi * 2;
Pi = 3.1398; // ошибка
```

В первой строке я объявил константу, которая будет равна 3.14. В следующей строке я использую ее в коде для получения двойного значения π , и результат сохраняется в простую переменную. А вот третья строка завершится ошибкой, потому что изменять константу нельзя.

Когда можно задействовать константы? Всегда, когда нужно использовать какое-то значение, которое, по вашему мнению, не должно изменяться во времени, — например, то же число π . Если вы просто будете в коде писать число 3.14, то ничего страшного в этом нет. Однако представьте, что, создав большой проект из тысяч строк кода, вы вдруг выясните, что нужно было использовать более точное значение π — например, с 10 знаками после запятой. Это приведет к тому, что придется просматривать весь код и исправлять все обращения к числу 3.14.

Тут кто-то скажет, что для этого есть операция поиска и замены, но может случиться так, что поиск/замена изменят не то, что нужно. Число 3.14 само по себе уникально, и у вас, скорее всего, с его поиском и заменой проблем не возникнет.

А если нужно изменить число 1000 в определенных ситуациях на число 2000, то в этом случае уже возникает большая вероятность случайной замены не в том месте.

Значение константы не просто желательно указывать сразу — оно должно быть тотчас определено. Значение должно быть известно уже на этапе компиляции, потому что в этот момент компилятор как раз заменяет все константы на их значения, т. е. меняет имя на значение.

2.10. Нулевые значения

Ссылочные переменные могут принимать *нулевое* значение: `null`. Когда вы присваиваете `null`, то это как бы указывает на то, что переменная уже ни на что не ссылается. Если все переменные, которые ссылались на память, уничтожены, сборщик мусора может освобождать память.

Нулевое значение иногда оказывается удобным при работе со ссылочными типами. Например:

```
string param = null;
...
// здесь может быть какой-то код
...
if (param == null)
    param = "значение не было определено";
else
    param = "значение где-то определили";
```

Если переменной не присвоено значение, мы не можем к ней обращаться и даже использовать оператор сравнения. Но если мы присвоим ей значение `null`, то это уже хоть какое-то значение, переменная становится рабочей, и мы можем использовать сравнение.

Простые типы не могут принимать значение `null`, но мы можем сделать так, чтобы это стало возможным. Надо просто после имени типа данных указать вопросительный знак. Например:

```
int? i = null;
```

В этом примере переменная `i` объявлена как целочисленная, а вопросительный знак после имени типа указывает на то, что она может принимать значение `null`. Именно это я и демонстрирую в приведенном примере.

2.11. Начальные значения переменных

Допустим, мы объявили переменную — какое значение она получит? Интересный вопрос, и тут все зависит от того, где объявлена переменная. Если это переменная класса, то действуют следующие правила:

- ❑ числовые переменные инициализируются нулем;
- ❑ переменные типа `char` тоже равны нулю: `'\0'`;

- булевы переменные инициализируются в `false`;
- объекты остаются неинициализированными и получают значение `null`.

А что происходит со строковыми переменными? Строки в C# — это тоже объекты, а значит, по умолчанию такие переменные будут равны `null`. Строки, как и объекты, нужно инициализировать явно. Единственное различие — для их инициализации можно использовать простое присваивание строкового значения.

Если переменная объявлена локально для какого-то метода, то вне зависимости от типа она не получает никаких начальных значений и ее нужно будет инициализировать явно. Компилятор просто не даст использовать переменную без инициализации и выдаст ошибку. Следующий пример не будет откомпилирован:

```
int foo()
{
    int i = 0;    // все в порядке
    int j;
    return i + j; // ошибка, обращение к неинициализированной j
}
```

2.12. Системы счисления

Человек привык работать с десятичной системой счисления, когда каждый разряд в числе является степенью 10. Посмотрим на число 345. Мы сейчас не вникаем в то, что означает это число, но в моем детстве в школе мы разбирали его по составляющим. Здесь у нас 5 единиц, 4 десятка и 3 сотни. А что такое единицы, десятки и сотни? Это степени числа 10:

- единицы = 10 в нулевой степени, а десять в нулевой — это 1;
- десятки = 10 в первой степени, это 10;
- сотни = 10 во второй степени, это 100.

Чтобы разобрать число 345 в десятичной системе мы должны делить его составляющие на 10. Берем 345, делим на 10, получаем 34 и 5 остаток. Записали просто 5. Теперь делим 34 на 5, получаем 3 и 4 в остатке. Затем 3 делим на 10 — получим 0 и 3 идет в остаток. Таким образом мы получили три числа: 3, 4 и 5, если просто записать их в порядке, обратном получению.

Теперь попробуем собрать эти три числа: 3, 4 и 5. Это $3 \cdot 10^0 + 4 \cdot 10^1 + 5 \cdot 10^2$. Упрощаем: $3 \cdot 1 + 4 \cdot 10 + 5 \cdot 100$.

Получается, что мы должны взять каждую цифру числа справа налево и умножить ее на 10 в той степени, цифру которой мы берем, нумеруя их с нуля.

Точно так же работает абсолютно любая система счисления. Не удивлюсь, что если бы в детстве мы учились на восьмеричной системе, то для нас она была бы привычнее.

Но вот компьютер — это электроника, которая работает от электричества, и эту «железку» с детства научили работать с двоичной системой. Ей это проще, потому

что в двоичной системе только два состояния: включено или выключено, есть ток или нет тока.

Если в десятичной системе счисления используются десять цифр: от 0 до 9 (опять нумерация с нуля), то в двоичной системе только два числа: 0 и 1. Сколько их будет в восьмеричной? Ответ прост, восемь: от 0 до 7. Какая система счисления, столько различных цифр и используется.

В десятичной системе мы делили число на 10 и потом брали остаток, в двоичной же системе мы должны делить на 2.

Возьмем число 10 в десятичной системе и разберем его:

1. Делим сначала 10 на 2. Результат 5, остатка нет, значит, записываем 0:

0

2. Теперь делим результат 5 на 2 — будет 4 и остаток 1:

01

3. Теперь делим 4 на 2 — будет 2 и остаток 0:

010

4. Последнее число 2 делится на 2 — получается 1. Остатка нет, так что эту единицу добавляем к результату:

0101

5. Последний штрих. Здесь, как и в случае с десятичной системой, мы получали числа в обратном порядке, значит, число нужно развернуть:

1010

Отлично, у нас получилось двоичное выражение числа 10. Теперь, чтобы собрать его обратно, нужно каждую цифру умножить на 2 в степени позиции цифры и сложить результат справа налево:

$$0 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 1 \cdot 2^3$$

Полужирным шрифтом я выделил здесь оригинальное число. Упростим этот вид записи расчета степени двойки:

$$0 \cdot 1 + 1 \cdot 2 + 0 \cdot 4 + 1 \cdot 8$$

Здесь четыре слагаемых, где первое и третье — это умножение на 0, которое дает 0:

$$1 \cdot 2 + 1 \cdot 8$$

Если сначала делать умножение, а потом сложение, то получается 10 — мы вернулись в исходное состояние.

Попробуем разобрать и собрать число 22:

1. Делим его на 2 — результат 11, остаток ноль:

0

2. Делим результат 11 на 2 — результат 5, остаток 1, записываем:

01

3. Результат 5 делим на 2 — результатом будет 4, остаток 1, записываем:
011
4. Результат 4 делим на 2 — результатом будет 2, остаток 0, записываем:
0110
5. И последний результат делим на 2 — будет 1, его и записываем:
01101
6. Остается снова развернуть это число:
10110

Собирать опять же нужно в обратном порядке (я здесь попробовал записать число чуть по-другому — может, так будет нагляднее):

$$\begin{aligned} &0 \cdot 2^0 + \\ &1 \cdot 2^1 + \\ &1 \cdot 2^2 + \\ &0 \cdot 2^3 + \\ &1 \cdot 2^4 \end{aligned}$$

Расчитаем степени:

$$\begin{aligned} &0 \cdot 1 + \\ &1 \cdot 2 + \\ &1 \cdot 4 + \\ &0 \cdot 8 + \\ &1 \cdot 16 \end{aligned}$$

В первой и четвертой строках производится умножение на 0, так что эти части обнуляются. Остается: $1 \cdot 2 + 1 \cdot 4 + 1 \cdot 16$. И опять же, если сначала делать умножение, а потом сложение, то результат — 22.

Когда мы объявляем переменные, то можем указывать значения как в десятичной системе, так и в двоичной. Чтобы записать значение в двоичной системе, нужно просто в начале этого значения поставить символы `0b`, где буква `b` означает `binary` (двоичная). Вот так можно в двоичной системе завести числовую переменную и присвоить ей число 22:

```
int b = 0b00010110;  
Console.WriteLine(b);
```

В результате выполнения этого кода в консоли мы должны увидеть число 22.

Чтобы было удобнее читать, в двоичной (бинарной) системе принято записывать число блоками по четыре цифры и разделять на блоки символами подчеркивания:

```
int b = 0b_0001_0110;
```

Кстати, десятичные числа тоже можно записывать блоками, разделяя подчеркиваниями. Например, число 123456789 можно записать так:

```
int b = 123_456_789;
```

Намного же удобнее читать: 123 миллиона, 456 тысяч 789.

Очень часто в ИТ применяется еще и шестнадцатеричная система счисления, где можно увидеть 16 цифр (от 0 до 15). Как так, их же всего 9! Это в привычной нам десятичной. А в компьютерной науке для обозначения чисел в 16-ричной системе используются привычные нам цифры от 0 до 9, а потом идут А (это 10), В (это 11), С (это 12), D (это 13), Е (это 14) и F (это 15). То есть латинские буквы от А до F заменяют цифры от 10 до 15.

Давайте возьмем число 28 и переведем его в 16-ричную систему (я взял число поменьше, чтобы было проще сразу видеть результат). Итак, делим 28 на 16 — результатом будет 1 и остаток 12. Чтобы записать 12 в виде числа, используем букву С. Теперь остаток 1 просто добавляем к С и получаем 1С.

Теперь вернем 1С обратно в десятичную систему:

$$С \cdot 16^0 + \\ 1 \cdot 16^1$$

Или $С \cdot 1 + 1 \cdot 16$. С — это 12, а значит, $12 \cdot 1 + 1 \cdot 16$, и мы вернулись к 28.

Надеюсь, идея понятна. Таким образом можно работать с любой системой счисления.

В коде мы можем напрямую использовать шестнадцатеричные числа — для их записи вначале нужно добавить символы 0x. То есть число 28 в шестнадцатеричной системе мы могли бы присвоить напрямую:

```
int i = 0x1C;
Console.WriteLine(i);
```

2.13. Логические операции

В программировании используется не только привычная нам математика: сложение, вычитание, деление, умножение, но и интересные операции: И, ИЛИ, Исключающее ИЛИ.

Операцию ИЛИ называют *логическим сложением*, операция И — это *логическое умножение*. Исключающее ИЛИ (XOR) — это *логическое вычитание*.

Логические операции часто показывают на логическом типе данных, а в C# это Boolean. У логического типа два состояния: true или false. Но точно так же можно было показывать это и на числах: 0 и 1, как я делал для двоичной системы счисления.

В табл. 2.2 показаны в первых двух колонках варианты значения, над которыми производятся операции, а в остальных колонках — результат, соответствующий операции.

Таблица 2.2. Лoгические операции

Значение 1	Значение 2	И	ИЛИ	Исключающее ИЛИ
True	True	True	True	False
True	False	False	True	True
False	True	False	True	True
False	False	False	False	False

- ❑ Операция И — достаточно простая: она дает истину, если оба значения равны истине. Во всех остальных случаях это ложь. Если хотя бы одно значение говорит ложь, то мы воспринимаем результат как ложный.
- ❑ Операция ИЛИ будет давать правду, если хотя бы одно значение говорит правду. Ложью будет только, если все говорят ложь.
- ❑ Исключающее ИЛИ заключается в том, что результатом будет правда, только если есть разные значения правды и лжи. Если все говорят правду или все говорят ложь, то результатом будет ложь.

Возьмем два числа в бинарной системе: 6 и 16 — и посмотрим, что будет в результате лoгического сложения:

```
int a = 0b_0000_0110;  
int b = 0b_0001_0010;  
Console.WriteLine(a | b);  
int c = 0b_0001_0110;
```

Мы получили 22. В конце этого примера я завел для наглядности еще одну переменную (c) и в ней указал результат — это число 22 в бинарной системе. Посмотрите на числа a, b и c. У числа c единицы в тех местах, где у a или у b есть единицы: или у a, или у b.

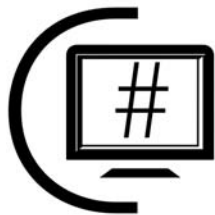
А теперь посмотрим на лoгическое И двух следующих чисел:

```
int a = 0b_0000_0110;  
int b = 0b_0001_0100;  
Console.WriteLine(a & b);  
int c = 0b_0000_0100;
```

Результатом будет 4, потому что только в этом бите у переменных a и b есть единицы.

С бинарной математикой программисты в наше время сталкиваются не так часто, как это было много лет назад. Обычно с ней приходится работать, когда используются флаги, о которых речь пойдет в разд. 5.3.2. Почему не рассмотреть их сейчас, или почему бы мне не рассказать о лoгических операциях там? Суть моего подхода в том, что вы сейчас прочитали про лoгические операции, сделаете перерыв, а потом мы снова начнем о них говорить, и если что-то вылетит/забудется, то придется вернуться и повторить материал. Повторение — мать учения, и я буду возвращаться к уже пройденному материалу и обращаться к нему еще не раз.

ГЛАВА 3



Объектно-ориентированное программирование

К этому моменту мы уже обрели серьезные базовые знания, и полученной нами информации лет 30 назад было бы достаточно для написания программ на уровне того времени. В общем-то, мы тоже можем сейчас что-то написать в консоли, но кого удивишь текстовым интерфейсом в наши годы господства интерфейса визуального? Нечто более сложное писать в линейном режиме весьма трудоемко, поэтому и была предложена концепция *объектно-ориентированного программирования* (ООП), ставшая основой построения современных языков.

В этой главе с концепцией объектно-ориентированного программирования мы и познакомимся.

3.1. Объекты на C#

Давайте посмотрим, что нам приготовил простейший C#-проект без использования операторов верхнего уровня. Такой проект мы создали в *разд. 1.3*. Напомню код, который был сгенерирован при создании проекта:

```
class EasyCSharp
{
    public static void Main()
    {
        Console.WriteLine("Hello, World!!!");
    }
}
```

C# — это полностью объектный язык, и в нем все построено вокруг понятий *класс* и *объект*. Что такое класс? Я бы назвал его самодостаточным блоком кода, имеющим все необходимые свойства и методы. Хороший класс должен обладать логической целостностью и завершенностью, чего далеко не всегда удается достичь начинающим программистам, но все приходит с опытом. Хороший класс должен решать одну задачу, а не несколько сразу.

Я люблю объяснять принципы объектной технологии на примере строительных сооружений. Так вот, допустим, что сарай — это объект. Он обладает *свойствами*:

высотой, шириной и глубиной — ведь по своей форме сарай напоминает параллелограмм (мы берем самый простой вариант), для описания которого нужно знать длину его трех граней. А сараев может быть в городе много, все они обладают одними и теми же свойствами, и чтобы проще было обращаться ко всем сараям сразу, можно ввести для них понятие класса.

То есть класс — это описание объекта, так сказать, его проектная документация или чертеж. По одному классу можно создать несколько объектов. Попробуем создать описание такого объекта, т. е. описать класс объекта «сарай»:

```
Класс Сарай
Начало описания
    Число Высота;
    Число Ширина;
    Число Глубина;
    Строка Название;
Конец описания
```

Получилось, что у нашего объекта есть четыре свойства. Для описания трех из них нужны числа — ведь эти параметры имеют метрическую природу. Так, в свойстве высоты может быть число — например, 5 метров, но никак не может быть слова. Однако слово может присутствовать в свойстве названия сарая.

У объекта могут быть и *методы*, т. е. какие-то присущие ему действия. Какой бы метод дать сараю? Ничего жизненного не приходит на ум, потому что сарай сам по себе ничего делать не умеет, он просто стоит. Поэтому давайте наделим его возможностью говорить нам о своих размерах. Это, пожалуй, пример из области фантастики — впрочем, на сарае может же быть табличка с информацией о его размерах. Хотя нет, табличка — это информация статичная, и это тоже свойство.

Ладно, поступим по-другому — допустим, что на сарае есть кнопка, по нажатию на которую из окошка вылетает птичка и сообщает нам размеры сарая. Боже, какой бред я несу...

Откуда птичка узнает эти размеры? Она должна спросить их у сарая, а сарай ей это скажет. Честное слово, не пил я сегодня... Ладно, продолжим в том же духе и назовем этот метод: *ПолучитьОбъем()*.

Тогда описание объекта будет выглядеть следующим образом:

```
Объект Сарай
Начало описания
    Число Высота;
    Число Ширина;
    Число Глубина;
    Число ПолучитьОбъем()
Начало метода
    Подсчитать объем сарая
Конец метода
Конец описания
```

В этом примере мы наделили объект методом *ПолучитьОбъем()*. Перед именем метода указывается тип значения, которое может вернуть метод. В нашем случае он вычисляет объем, который является числом. После имени метода указываются круглые скобки.

Примерно так же описание объектов осуществляется и на языке C#. Все начинается с ключевого слова `class`, которое говорит о начале описания объекта. Напоминаю, что класс — это описание объекта, а объект — это *экземпляр*, созданный на основе этого класса. После слова `class` идет имя класса. Давайте вспомним пустой класс, который нам создал мастер при формировании пустого приложения:

```
using System;

namespace HelloWorld
{
    class EasyCSharp
    {
        public static void Main()
        {
        }
    }
}
```

Мастер создал нам класс с именем `EasyCSharp`, у которого есть один метод: `Main()`. Этот метод имеет определенное значение — он всегда при запуске приложения выполняется первым. Именно поэтому мы писали в этом методе свой код и видели его в консоли. Кое-что начинает вырисовываться? Если нет, то ничего страшного, скоро каждое слово встанет на свое место, и все будет понятно.

Описание класса заключено в пространство имен. Как я отмечал в *разд. 2.3*, чтобы избежать конфликтов, лучше всего всё помещать в свои пространства имен. Там же я говорил о пространствах имен уровня файла, когда в начале файла задается имя пространства, и оно распространяется на весь файл. Следующий код будет *почти* идентичен предыдущему:

```
using System;

namespace HelloWorld;

class EasyCSharp
{
    public static void Main()
    {
    }
}
```

Разница между ними заключается в том, что после имени пространства имен идет точка с запятой, а не фигурные скобки, и пространство в этом случае будет распространяться на весь файл.

Теперь давайте начнем описывать класс «сарай». В нашем случае объявляется новый класс `Shed` (в следующих примерах кода я буду опускать имя пространства имен, чтобы сэкономить место):

```
namespace MyTests;
```

```
class Shed  
{  
}
```

Между фигурными скобками будет идти описание класса — та самая его проектная документация. Поскольку класс — это только проектная документация, то для работы нужно сначала создать объект на основе класса. Но для этого надо объявить переменную. Переменная типа класса создается точно так же, как и другие переменные. Например, переменная типа сарая может быть объявлена следующим образом:

```
Shed myFirstShed;
```

Теперь переменную нужно проинициализировать, ибо она пустая, и система не сможет определить необходимую для нее память. Никаких ассоциаций не возникает? В случае с массивами мы тоже не знали нужного размера, и для их инициализации должны были использовать оператор `new`. Здесь та же песня с тем же бубном. Вот так будет выглядеть создание объекта из класса:

```
myFirstShed = new Shed();
```

Здесь переменной `myFirstShed`, которая имеет тип `Shed`, присваивается результат работы оператора `new`, а после него пишется имя класса, экземпляра которого мы создаем, и ставятся круглые скобки. Пока не нужно задумываться, почему в этом месте стоят круглые скобки, — просто запомните, что они тут обязательны.

Конечно же, объявление и инициализацию можно записать в одну строку, и именно так мы будем делать чаще всего ради экономии места, да и просто потому, что такой код выглядит эстетичнее. Но иногда объявление может стоять отдельно от инициализации.

Из одного класса вы можете создать несколько объектов, и каждый из них будет являться самостоятельной единицей:

```
Shed myFirstShed = new Shed();  
Shed mySecondShed = new Shed();
```

Здесь объявлены и тут же проинициализированы уже два объекта. Объекты имеют не только разные имена, но могут обладать разными значениями свойств, т. е. у этих сараев могут быть разные размеры. Мы пока не наделили их такими свойствами, но скоро сделаем это.

В .NET есть пять *модификаторов доступа*, с помощью которых мы можем указать, как будет использоваться метод, свойство или сам класс:

- ❑ `public` — член объекта (метод или свойство) доступен всем;
- ❑ `protected` — член объекта доступен только самому объекту и его потомкам;
- ❑ `private` — член объекта является закрытым и не доступен за его пределами и даже для потомков. Если ни один модификатор не указан, то будет использоваться `private`;
- ❑ `internal` — член доступен только в пределах текущей сборки;
- ❑ `protected internal` — доступен всем из текущей сборки, а также типам, производным от текущего, т. е. перед нами что-то вроде объединения модификаторов доступа `protected` и `internal`.

Модификаторы доступа можно использовать со свойствами, методами и даже с простыми переменными, являющимися членами класса, хотя последнее нежелательно — лучше все же использовать свойства, а не простые переменные, но о свойствах мы будем говорить в *разд. 3.2*.

Модификаторы доступа могут быть установлены не только членам класса, но и самим классам. Открытые (`public`) классы доступны для всех, вне зависимости от сборки и места жительства. Классы `internal` доступны только внутри определенной сборки. Модификаторы `private` и `protected` могут использоваться только со вложенными классами, поэтому мы рассмотрим их отдельно.

ВНИМАНИЕ!

Любые переменные, свойства, методы и даже классы, которым явно не указан модификатор доступа, по умолчанию получают модификатор `private`.

3.2. Свойства

Как мы уже поняли, объект может обладать *свойствами*. В некоторых объектных языках свойства — это просто переменные, которые принадлежат классу. В C# свойства отличаются от переменных и являются отдельной структурой данных.

Давайте расширим наш класс *сарая* и добавим в него свойства. Для начала объявим в классе две переменные:

```
class Shed
{
    int width;
    int height;
}
```

Переменная, которая объявлена в классе, называется *полем*. Я буду использовать как термин *переменная класса*, так и *поле*, но это одно и то же.

Мы объявили две переменные внутри фигурных скобок описания класса. Пока что это всего лишь переменные, а не свойства, и к ним невозможно получить доступ извне, потому что по умолчанию (отсутствуют модификаторы доступа) переменные и методы создаются закрытыми и извне недоступны. Да, мы могли бы написать так:

```
class Shed
{
    public int width;
    public int height;
}
```

Теперь переменные открыты, но не делайте так. Несмотря на то что такая запись возможна — это плохое программирование. Переменные должны оставаться закрытыми (`private` или `protected`), а вот чтобы сделать их открытыми, нужно объявить их свойства следующим образом:

```
int width;

public int Width
{
    get { return width; }
    set { width = value; }
}
```

Здесь элемент `width` записан с маленькой (строчной) буквы — это переменная, которая становится полем, т. е. переменная, объявленная внутри класса. Я обычно их с маленькой буквы и пишу. Потом идет свойство `Width` с большой (прописной) буквы. Учитывая, что C# чувствителен к регистру букв, такое объявление легально, и мы можем иметь и переменную, и свойство с одним именем, но хотя бы одна буква в имени должна быть в другом регистре.

Свойства обычно создаются для того, чтобы они были открытыми, поэтому объявление начинается с модификатора доступа `public`. После этого идет тип данных и имя. В .NET принято именовать переменные с маленькой буквы, а соответствующие свойства — с большой. Некоторые любят начинать переменные с символа подчеркивания или добавлять `m_` (в нашем случае это было бы `m_width`, где приставка `m` означает *member*). Это тоже нормально, главное — именовать одинаково. В отличие от объявления переменной, тут нет в конце имени точки с запятой, а открываются фигурные скобки, внутри которых нужно реализовать два *аксессуара* (*accessor*): `get` и `set`. Аксессуары позволяют указать доступ к свойству на чтение (`get`) и запись (`set`). После каждого аксессуара в фигурных скобках указываются также действия свойства.

Для `get` нужно в фигурных скобках выполнить оператор `return` и после него указать, какое значение должно возвращать свойство. В нашем случае свойство возвращает значение переменной `width`, т. е. при обращении к свойству `Width` мы будем видеть значение переменной `width`.

Для `set` мы должны сохранить в какой-то переменной значение свойства, которое хочет установить пользователь. Значение свойства находится в виртуальной переменной `value`. Почему виртуальной? Потому что мы нигде ее не объявляли — она существует всегда внутри фигурных скобок после ключевого слова `set` и имеет такой же тип данных, что и свойство. В нашем примере при изменении свойства значение сохраняется в переменной `width`.

Подведем итог: при изменении свойства значение попадает в переменную `width`, и при чтении свойства мы получаем значение из той же переменной.

Переменным можно задавать значения сразу же при объявлении. С полями все точно так же — можно не просто завести поле `width`, а сразу же сказать, что оно будет равно 10:

```
int width = 10;

public int Width
{
    get { return width; }
    set { width = value; }
}
```

Получается, что в нашем случае свойство просто является оберткой для переменной объекта. Зачем нужна эта обертка? — ведь можно было просто открыть доступ к переменной и обращаться к ней напрямую... Можно, но не нужно, потому что это нарушает принципы безопасности. Что здесь обеспечивает безопасность? Рассмотрим различные варианты — например, следующее свойство:

```
public int Width
{
    get { return width; }
}
```

Здесь объявлено свойство `Width`, у которого есть только аксессор `get`. Что это может значить? Мы можем только прочесть переменную `width` через свойство `Width`, но не можем ее изменить. Получается, что мы создали свойство, которое доступно только для чтения.

А вот еще пример:

```
public int Width
{
    get { return width; }
    set
    {
        if (value > 0 && value < 100)
            width = value;
    }
}
```

Здесь возвращаемое свойством значение осталось прежним, зато код изменения свойства претерпел существенную переработку — добавлена проверка на то, чтобы ширина сарая не была отрицательной или нулевой и была менее 100 (слишком большие сараи мы не строим). Вот таким образом свойства могут защищать переменные класса.

А что если у вас множество переменных, которые не нуждаются в защите, и им надо всего лишь создать обертку? Писать столь большое количество кода доста-

точно накладно. Тут можно использовать *сокращенное объявление свойства*. Давайте сокращенным методом объявим глубину сарая:

```
public int Lengthwise { get; set; }
```

В фигурных скобках после ключевых слов `get` и `set` нет никакого кода, а сразу стоят точки с запятой. Что это значит? Все очень просто — свойство `Lengthwise` будет являться одновременно свойством для внешних источников и переменной для внутреннего использования. Конечно же, у нас нет кода, а значит, мы не можем (и просто некуда) добавлять код защиты, поэтому такой метод используется там, где свойства являются просто оберткой для переменной.

Теперь посмотрим, как можно использовать свойства класса, т. е. изменять их или читать значения. Мы уже знаем, как объявлять объект определенного класса (нужно объявить переменную этого класса) и как его инициализировать. Когда объект проинициализирован, вы можете использовать его свойства, читать их и изменять. В общем виде доступ к свойству записывается следующим образом:

Имя_Объекта.Свойство

Одно важное замечание: слева от точки указывается именно имя объекта, а не класса, т. е. имя определенного экземпляра класса. Через класс можно обращаться к статичным переменным, но это другая песня, о которой будет отдельный разговор.

Следующий пример показывает, как можно использовать свойство `Height` нашего сарая:

```
Shed myFirstShed = new Shed();  
myFirstShed.Width = 20;  
myFirstShed.Height = 10;  
int h = myFirstShed.Height;
```

В первой строке мы объявляем и инициализируем объект `myFirstShed` класса `Shed`. Во второй строке ширина меняется на 20. Помните я говорил, что при объявлении поля мы можем задать значение по умолчанию для поля, и тогда мы указали 10. Здесь мы изменили его на 20. В третьей строке высоте сарая устанавливаем значение 10. В последней строке кода мы уже читаем установленное значение и сохраняем его в новой переменной `h`.

В листинге 3.1 показан полноценный пример, который объявляет класс сарая и показывает, как его можно было бы использовать. Обратите внимание, что при объявлении объекта пространство имен опущено, потому что класс `Shed` объявлен и используется внутри одного и того же пространства имен: `PropertiesExample`.

Листинг 3.1. Работа со свойствами класса

```
using System;  
using System.Text;  
  
namespace PropertiesExample
```

```
{
    // Объявление класса программы
    class Program
    {
        static void Main(string[] args)
        {
            // Создаем объект
            Shed myFirstShed = new Shed();
            // Задаем значения свойств
            myFirstShed.Height = 10;
            myFirstShed.Width = 20;
            myFirstShed.Lengthwise = myFirstShed.Width;

            // Вывод на экран значений
            Console.WriteLine("Высота: " + myFirstShed.Height);
            Console.WriteLine("Ширина: " + myFirstShed.Width);
            Console.WriteLine("Глубина: " + myFirstShed.Lengthwise);

            Console.ReadLine();
        }
    }

    // Класс сарая
    class Shed
    {
        int width;
        public int Width
        {
            get { return width; }
            set { width = value; }
        }

        int height = 10;
        public int Height
        {
            get { return height; }
            set { height = value; }
        }

        public int Lengthwise { get; set; }
    }
}
```

В методе `Main()` класса `Program` мы объявляем переменную типа `Shed` с именем `myFirstShed`. Затем создаем экземпляр класса `Shed` (объект класса `Shed`) и сохраняем его в переменной `myFirstShed`. Теперь мы можем назначить свойствам объекта числовые значения (размеры сарая).

ПРИМЕЧАНИЕ

Исходный код этого примера можно найти в папке *Source\Chapter3\Properties* сопровождающего книгу электронного архива (см. приложение).

Аксессоры `get` и `set` могут иметь модификаторы доступа. По умолчанию аксессоры создаются открытыми (`public`) для общего использования. Если нужно сделать так, чтобы свойство нельзя было изменить, аксессор `set` можно объявить как `private`:

```
public int Width
{
    get { return width; }
    private set { width = value; }
}
```

или — в сокращенном варианте:

```
public int Width
{
    get;
    private set;
}
```

Поскольку аксессор `set` объявлен здесь как закрытый, свойство не может быть изменено извне, потому что к нему нет доступа. Получается, что это еще один метод создать свойство только для чтения, однако в этом случае вы можете получить доступ к свойству на запись внутри текущего класса, — запрет распространяется лишь на внешний доступ.

Если вы обращаетесь к свойству или к переменной класса из внешнего мира (из другого класса), то его имя тоже нужно писать полностью:

Имя_Объекта.Свойство

В C# 6 задавать значения по умолчанию для свойств, которые не используют явной переменной класса, стало намного проще — достаточно после объявления просто присвоить это значение:

```
public int Width { get; set; } = 10;
```

В последнее время C# движется все больше в сторону безопасности, и если мы добавим свойство для хранения свойства:

```
public string Name { get; set; }
```

то Visual Studio выделит `Name` зеленым цветом. Дело в том, что мы объявили свойство, но не задали ему значение по умолчанию. То есть сразу после создания объекта сара я имя будет пустым и равно `null`, и среда разработки предупреждает нас об этом.

Есть три варианта решения этой проблемы:

- явно указать, что имя может принимать `null` значения:

```
public string? Name { get; set; }
```

- задать значение по умолчанию:

```
public string Name { get; set; } = "Неизвестно";
```

- и третий вариант — заявить, что мы гарантируем, что имя будет задано после создания:

```
public string Name { get; set; } = null!;
```

В любом из этих случаев среда разработки больше не будет как-то выделять имя свойства.

3.3. Методы

У нашего объекта `EasyCSharp`, который мы рассмотрели в *разд. 1.3.1*, нет свойств, а есть только один (основной) метод `Main()`:

```
public static void Main()
{
    Console.WriteLine("Hello, World!!!");
}
```

О том, что это *метод*, говорят круглые скобки после имени, а перед именем указывается тип метода. В нашем случае тип значения, которое возвращает метод, равен `public static void` (открытый, статичный, пустой). Пока рассмотрим только ключевое слово `void`, которое означает, что возвращаемого значения нет.

После имени метода в круглых скобках могут передаваться *параметры*. В нашем случае скобки пустые — это означает, что никаких параметров нет.

В фигурных скобках записывается код (действия), который выполняет этот метод. В нашем случае в качестве действия выполняется строка:

```
Console.WriteLine("Hello, World!!!");
```

Эта строка не что иное, как вызов метода другого класса. Вызов методов класса (статичных методов) происходит следующим образом:

```
Имя_Класса.Имя_Метода (Параметры);
```

Если методы класса вызываются через объект, то статичные методы — через класс. Подробнее о статичных методах и свойствах будет рассказано в *разд. 3.3.5*, а пока же нас интересует формат вызова. Далее мы будем обсуждать простые нестатичные методы, если явно не указано, что метод статичный.

Если метод находится в том же объекте, из которого мы его вызываем, то вызов можно сократить до:

```
Имя_Метода (Параметры);
```

Таким образом, метод `WriteLine()` выводит в окно консоли указанное текстовое сообщение.

Если вы обращаетесь к свойству или к переменной класса из внешнего мира, то имя свойства или переменной тоже надо писать полностью:

```
Имя_Объекта.Имя_Свойства;
```

Но если мы обращаемся к свойству класса из метода, принадлежащего этому же классу, то нужно писать просто имя свойства или переменной без указания переменной объекта.

Возможно, я дал слишком много сухой информации про методы, но практика расставит всё по своим местам, — далее в этой главе мы будем практиковаться.

Метод `Main()` является основным, и именно с него программа начнет свое выполнение. Это значит, что хотя бы один класс должен содержать метод с таким именем, иначе будет непонятно, откуда программа должна начать выполняться. Этот метод не обязательно должен быть где-то в начале файла — он может находиться в любом месте, потому что выполнение программы осуществляется не от начала файла к концу, а начиная с `Main()`, а дальше — как жизнь прикажет.

3.3.1. Описание методов

Теперь научимся создавать простейшие методы и использовать их. В общем виде объявление метода записывается следующим образом:

```
модификаторы значение Имя(параметры через запятую)
{
    Код;
    return значение;
}
```

Модификаторами доступа могут выступать уже знакомые нам `public`, `protected` и `private`, с помощью которых мы можем определить, доступен ли метод внешним классам или наследникам, о которых мы пока еще не говорили. Мы уже знаем, что если метод не должен возвращать значения, то надо указать ключевое слово `void`. Если параметры методу не нужны, то в круглых скобках ничего указывать не следует.

В фигурных скобках мы можем написать код метода, а если метод должен возвращать значение, то его надо указать после ключевого слова `return`. Этот оператор может находиться в любом месте кода метода, хоть в самом начале, но нужно знать, что он прерывает работу метода и возвращает значение. Если метод не возвращает значение, а, точнее, возвращает `void`, то `return` необязателен, — разве что вы хотите прервать работу метода.

Тут, наверное, у вас в голове начинается каша, и пора перейти к практическим примерам, чтобы закрепить все сказанное в коде и увидеть методы в жизни, а также разобраться, для чего они нужны. В *разд. 3.2* мы создали небольшой, но очень красивый и удобный сарай, и даже хотели реализовать бред в виде вылетающей птички, которая сообщает размеры сарая. Пора этот бред превратить в реальность:

```
class Shed
{
    // здесь идет объявление переменных и свойств класса
    ...
    // метод возврата размера
    public int GetSize()
    {
        int size = width * height * Lengthwise;
        return size;
    }
}
```

Здесь я показал, что свойства объявлены до метода, но это не является обязательным требованием, — методы могут быть описаны вперемешку со свойствами. Но я привык отделять мух от котлет.

В примере объявлен метод с именем `GetSize()`. Причем объявлен как открытый: `public`, чтобы пролетающая мимо птичка могла спросить у сарая его размеры. Птичка не является частью сарая и не сможет увидеть его личные методы и свойства, но может увидеть открытые, поэтому метод, возвращающий размер, объявлен как `public`.

Размеры сарая мы задали как целые числа, хотя могли бы задать их и дробными. Тут нет особой разницы, какой указать тип, мы даже и не оговаривали размерность — что это: миллиметры, сантиметры или даже метры. Раз размеры — целые числа, то и объем сарая тоже будет целым числом, ведь объем — это перемножение ширины, глубины и высоты, а перемножение целых чисел всегда (по крайней мере, меня так учили в школе) дает результат в виде целого числа. В программировании есть еще такое понятие, как *переполнение* (превышение размера максимального числа), но это мы пока рассматривать не станем.

Методу `GetSize()` ничего не нужно передавать, потому что у него и так есть доступ ко всему необходимому для расчета в виде переменных класса.

Внутри метода в первой строке объявляется переменная `size`, и ей присваивается результат перемножения всех трех размерностей сарая. Во второй строке мы возвращаем результат вычисления с помощью ключевого слова `return`. Все то же самое можно выполнить в одной строке:

```
public int GetSize()
{
    return width * height * Lengthwise;
}
```

Здесь сразу после ключевого слова `return` мы в виде выражения показываем, что метод должен вернуть результат перемножения трех переменных. Последняя переменная `Lengthwise` является свойством. Помните, мы объявили это свойство в сокращенном варианте, без использования каких-либо переменных? То есть так в расчетах мы можем использовать не переменную, а свойство. В принципе, мы можем в расчете задействовать все три свойства:

```
public int GetSize()
{
    return Width * Height * Lengthwise;
}
```

Поскольку свойства ширины и высоты возвращают значения соответствующих переменных, то и нет разницы, что использовать в расчетах, но я бы предпочел именно переменные, а не свойства.

Теперь посмотрим, как можно использовать метод в коде:

```
static void Main(string[] args)
{
    Shed myFirstShed = new Shed();
    myFirstShed.Height = 10;
    myFirstShed.Width = 20;
    myFirstShed.Lengthwise = myFirstShed.Width;

    int size = myFirstShed.GetSize();
    Console.WriteLine("Объем: " + size);
    Console.WriteLine("Объем: " + myFirstShed.GetSize());

    Console.ReadLine();
}
```

Первые четыре строки не должны вызывать вопросов, потому что мы создаем объект `myFirstShed` класса `Shed` и сохраняем в свойствах объекта размеры. После этого начинается самое интересное: объявляется целочисленная переменная `size`, и ей присваивается результат выполнения метода. Вызов метода происходит почти так же, как и обращение к свойству, только в конце имени метода ставятся скобки, внутри которых по необходимости могут указываться параметры.

```
int size = myFirstShed.GetSize();
```

Итак, в общем виде вызов метода записывается следующим образом:

```
Имя_Объекта.Имя_Метода();
```

При этом, если нужно сохранить результат, вы можете присвоить переменной результат выполнения метода:

```
Переменная = Имя_Объекта.Имя_Метода();
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter3\Method` сопровождающего книгу электронного архива (см. приложение).

3.3.2. Параметры методов

Прежде чем мы двинемся дальше, давайте сделаем небольшое улучшение — вынесем код класса `Shed` в отдельный файл, чтобы с ним удобнее было работать и расширять. Для этого щелкните правой кнопкой мыши на имени проекта и в контекст-

ном меню выберите **Add | New Item**. В открывшемся окне в списке **Templates** выберите **Class** и в поле **Name** введите имя файла: `Shed.cs` (рис. 3.1). В C# принято давать файлам имена по имени класса, который в нем хранится, а каждый класс помещать в отдельный файл.

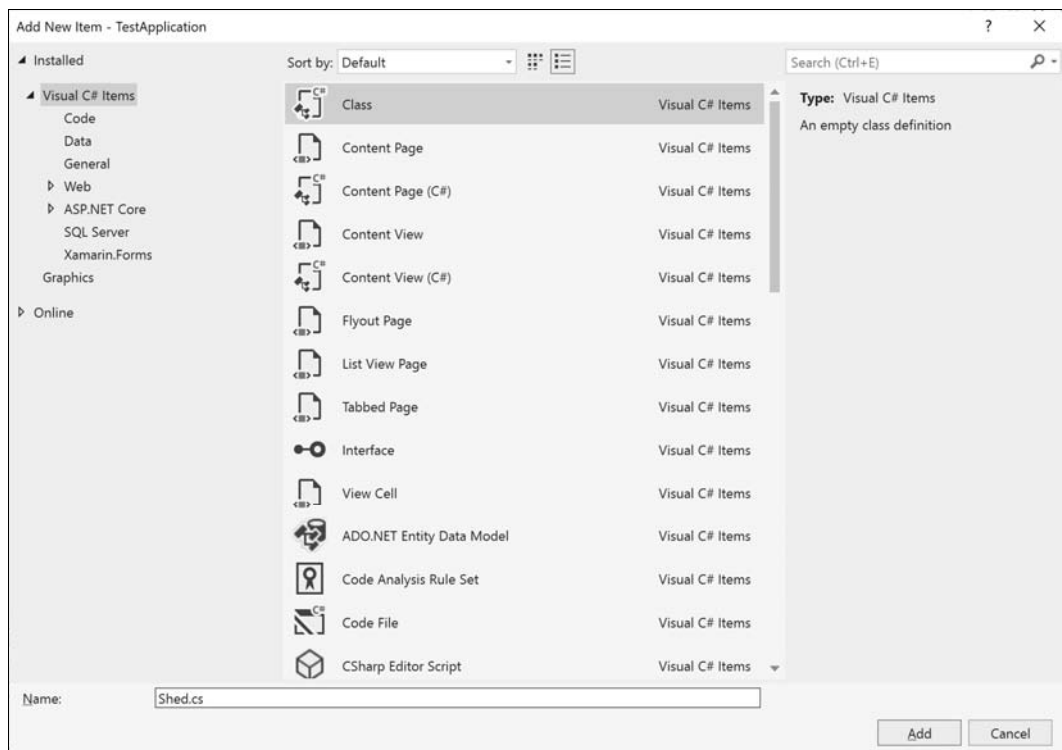


Рис. 3.1. Окно создания нового файла для кода

После нажатия кнопки **Add** новый файл будет добавлен в проект. Убедитесь, что в обоих файлах: `Shed.cs` и `Program.cs` — одинаковое пространство имен (namespace). Если оно одинаковое (а оно должно быть таким по умолчанию), то из файла `Program.cs` вы увидите все, что написано в таком же пространстве имен файла `Shed.cs`. Если имена пространств разные, то придется в файле `Program.cs` добавить в начало нужное пространство.

В файл `Shed.cs` внутрь пространства имен перенесите класс `Shed`, который мы написали в этой главе. В новом файле мастер Visual Studio уже поместил заготовку класса, которую можно удалить и заменить нашим классом. Теперь класс `Shed` находится в отдельном файле, и, на мой взгляд, с ним так удобнее будет работать. Пора обретать привычку оформлять проект аккуратно и правильно. Результат можно найти папке `Source\Chapter3\MethodParameters\` сопровождающего книгу электронного архива (см. приложение).

Постепенно мы движемся в сторону усложнения кода, и теперь давайте добавим в наш класс `Shed` метод, который будет увеличивать размеры сарая. А вдруг мимо

сарая пройдет бегемот и спросит у птички его размеры? Поняв, что он по габаритам в этот сарай не поместится, бегемот захочет расширить сарай, а для этого понадобится инструмент, в программировании выступающий методом. Метод этот мы назовем `ExpandSize()`, и он будет принимать три числовых параметра для хранения значений, на которые нужно увеличить сарай в ширину, высоту и глубину. Хочу опередить вас и ответить на вопрос, который, скорее всего, сейчас вертится в вашей голове: нет, я не курю, даже простые сигареты. Просто здания по своей природе статичны, и сложно придумать для них что-то реальное, что отвечало бы на вопрос: «Что делать?», т. е. выполняло действия.

Итак, метод расширения сарая может быть объявлен следующим образом:

```
class Shed
{
    // здесь свойства класса
    ...
    public void ExpandSize(int x, int y, int h)
    {
        Width += x;
        Lengthwise += y;
        Height += h;
    }
}
```

В скобках указываются три переменные и их типы — это будут параметры, которые нужно передать в таких же круглых скобках при вызове. Внутри кода банально на соответствующую величину из параметров увеличивается каждая из сторон сарая, чтобы он стал вместительнее.

Метод принадлежит классу `Shed`, поэтому объявлен внутри его описания (в дальнейшем я буду опускать код объявления класса и комментарии в стиле «здесь свойства класса», а стану сразу же указывать имя метода и говорить, какому классу он должен принадлежать). В ответ вы должны создать метод именно внутри указанного класса.

Теперь посмотрим, как вызывать метод:

```
Shed shed = new Shed();
shed.Height = 1;
shed.Width = 2;
shed.Lengthwise = 3;
shed.ExpandSize(2, 4, 6);
```

После создания класса и задания начальных значений вызываем метод `ExpandSize()`, передавая ему три числа, на значения которых должен увеличиться сарай, чтобы в него поместился бегемот. В качестве параметров можно было указывать не конкретные числа, а переменные, — главное, чтобы они соответствовали типам данных, указанным в объявлении метода, — в нашем случае это должны быть целочисленные переменные.

А что будет, если изменить значение передаваемого в метод параметра? Давайте проверим. Для этого напишем метод, который будет расширять размеры сарая и одновременно изменять переданные значения:

```
public void ExpandAndGetSize(int x, int y, int h)
{
    ExpandSize(x, y, h);
    x = Width;
    y = Lengthwise;
    h = Height;
}
```

Писать очередной код расширения сарая я не стал, а просто вызвал уже написанный ранее метод `ExpandSize()`. После этого я сохраняю в переданных переменных новые значения размеров. Как вы думаете, что произойдет? Давайте посмотрим и напишем вызов метода так:

```
shed.ExpandAndGetSize(2, 4, 6);
```

В качестве параметров мы передаем числовые значения. Компиляция пройдет успешно, и тут уже пора заподозрить неладное. Дело в том, что мы передаем в метод числа, а не переменные. Как же метод будет изменять эти числа? И как мы потом узнаем измененный результат? Да никак не узнаем, и он не изменится.

Все простые переменные по умолчанию передаются по значению. Что это значит? Метод получает не переменную и не память, где хранится значение, а копию значения. У метода автоматически создаются собственные переменные с именами, которые указаны в скобках метода, — в нашем случае это `x`, `y` и `h`. В эти переменные копируются переданные значения, и они больше никак не связаны с теми значениями, которые передавались. По завершении выполнения метода переменные уничтожаются, так что вы можете использовать `x`, `y` и `h` в своих расчетах и изменять их сколько угодно. В следующем примере внешние переменные `vx`, `vy` и `vz`, которые мы передаем в метод, никак не изменятся, потому что метод будет видеть не эти переменные, а значения, которые переданы:

```
int vx = 2;
int vy = 4;
int vz = 6;
shed.ExpandAndGetSize(vx, vy, vz);
```

А есть ли возможность передавать не значение, а именно переменную, чтобы внутри метода ее можно было изменять? Где это может пригодиться? Самый простейший случай — когда метод должен вернуть больше одного значения. Как можно вернуть не одно значение, а два? Если значения одного типа, то их можно вернуть в виде массива, а если разнотипные, придется идти уже на другие ухищрения, с которыми мы пока не знакомы.

Но есть еще один способ, который может быть удобен: один параметр вернуть в качестве возвращаемого значения, а другой — вернуть через один из параметров.

Для того чтобы в функцию передать не значение, а саму переменную, в объявлении метода перед именем параметра нужно указать ключевое слово `ref` (от Reference):

```
public void ExpandAndGetSize(ref int x, ref int y, ref int h)
{
    ExpandSize(x, y, h);
    x = Width;
    y = Lengthwise;
    h = Height;
}
```

Если переменная метода объявлена как `ref`, то при вызове нужно указывать именно переменную, а не значение (в нашем случае число). Теперь следующий вызов выдаст ошибку:

```
shed.ExpandAndGetSize(1, 2, 3);
```

Передавать нужно именно переменную, причем обязательно проинициализированную, потому что внутри метода будет передана ссылка на память переменной, а не значение, и метод станет работать уже с ее значением напрямую, а не через свою копию.

Помимо этого, перед каждой переменной нужно также поставить ключевое слово `ref`. Этим мы как бы подтверждаем, что передаем в метод ссылку:

```
int vx = 2;
int vy = 4;
int vz = 6;
shed.ExpandAndGetSize(ref vx, ref vy, ref vz);
Console.WriteLine("Размеры: " + vx + " " + vy + " " + vz);
```

Если переменная должна только возвращать значение, а внутри метода мы не станем использовать ее значение, то такой параметр можно объявить как `out`. Переменную, передаваемую как `out`, не обязательно инициализировать, потому что ее значение не будет использоваться внутри метода, но переменная, несущая измененное значение, обязательно будет там создана. Например, нам надо написать метод, первый параметр которого указывает, на сколько нужно увеличить каждую размерность сарая, а еще три параметра потребуются для возврата через них новых значений размеров сарая. Последние три параметра нет смысла объявлять как `ref`, вполне достаточно объявить их `out`, т. е. как только лишь выходное значение:

```
public void ExpandAndGetSize2(int inc,
    out int x, out int y, out int h)
{
    ExpandSize(inc, inc, inc);
    x = Width;
    y = Lengthwise;
    h = Height;
}
```

Теперь переменные `x`, `y` и `h` внутри метода во время вызова не будут содержать никаких значений, и если вы попытаетесь задействовать их, то компилятор сообщит об ошибке использования переменной, которой не назначено значение.

С другой стороны, вы должны задать таким переменным значения, которые будут возвращены коду, который вызвал метод:

```
public void ExpandAndGetSize2(int inc,
    out int x, out int y, out int h)
{
    Console.WriteLine(x);
}
```

Этот код не скомпилируется по двум причинам. Во-первых, переменная `x` не содержит никаких значений. Когда кто-то вызывает наш метод, то он предоставляет память для переменных, но их содержимое игнорируется, поэтому при компиляции произойдет ошибка — переменная `x` не определена. Во-вторых, в этом методе не задаются значения для переменных, которые объявлены как `out`.

А вот следующий код:

```
public void ExpandAndGetSize2(int inc,
    out int x, out int y, out int h)
{
    x = Width;
    y = Lengthwise;
    h = Height;
    Console.WriteLine(x);
}
```

уже скомпилируется корректно, потому что все три `out` параметра получают значения, и только потом мы используем переменную `x`, у которой значение задано в первой строке кода.

Поскольку первый параметр простой, а остальные три объявлены с ключевым словом `out`, использование метода будет выглядеть следующим образом:

```
shed.ExpandAndGetSize2(10, out vx, out vy, out vz);
Console.WriteLine("Размеры 2: " + vx + " " + vy + " " + vz);
```

Ключевое слово `out` ставится перед параметрами, которые станут выходными. После отработки метода переданные переменные будут содержать значения, которые были установлены внутри метода. Без ключевых слов `out` или `ref` это было бы невозможно.

А что если мы зададим значение для переменной `vs` и потом передадим эту переменную в метод как параметр `out`:

```
vs = 10;
shed.ExpandAndGetSize2(10, out vx, out vy, out vz);
```

Это можно, но бесполезно. При вызове методов с параметрами `out` значения этих переменных не передаются внутрь метода. Поэтому их и нельзя использовать внутри, и поэтому мы обязаны их проинициализировать внутри метода.

ПРИМЕЧАНИЕ

Исходный код этого примера можно найти в папке *Source\Chapter3\MethodParameters* сопровождающего книгу электронного архива (см. приложение).

Между ключевыми словами `out` и `ref` есть существенное различие. Если параметр объявлен как `out`, то его значение обязательно должно измениться внутри метода, иначе произойдет ошибка и программа может не скомпилироваться. Рассмотрим еще один более простой пример — метод `Sum()`, который не изменяет выходного параметра `result`:

```
static void Main(string[] args)
{
    int sum;
    Sum(1, 2, out sum);
}

static int Sum(int x, int y, out int result)
{
    return x + y;
}
```

В результате компиляции этого примера вы увидите ошибку: **The out parameter 'result' must be assigned to before control leaves the current method** (Выходной параметр 'result' должен быть назначен до того, как управление покинет метод). Проблему может решить следующий код:

```
static int Sum(int x, int y, out int result)
{
    result = x * y;
    return x + y;
}
```

Здесь переменная `result` уже изменяется на результат перемножения чисел. Таким образом, благодаря выходному параметру вызывающая сторона смогла получить два результата: сумму в качестве возвращаемого значения и произведение в качестве выходного параметра.

Ну а если параметр объявлен как `ref`, то его значение не обязано изменяться внутри метода.

Еще один модификатор, который может использоваться с параметрами методов, — `params`. Иногда бывает необходимо передать в метод переменное количество значений одинакового типа. В этом случае допускается передать значения в виде массива, который можно заполнить любым количеством цифр, а можно воспользоваться модификатором `params`. Для начала посмотрим, как это будет выглядеть в коде, а потом разберемся со всеми нюансами использования этого модификатора:

```
static int Sum2(params int[] values)
{
    int result = 0;
```

```
foreach (int value in values)
    result += value;
return result;
}
```

На первый взгляд, мы просто передаем массив значений и используем его в методе как массив. Зачем же тогда мы поставили у метода этот непонятный модификатор? Разница в его использовании заключается в способе вызова. Если параметр объявлен как `params`, то его значения при вызове просто приводятся через запятую, без создания какого бы то ни было массива. А поскольку у нас массив, то значений может быть переменное количество:

```
Sum2(1, 2, 3);
Sum2(1, 2, 3, 4, 5);
```

Оба вызова вполне корректны: среда объединит все значения в массив и передаст их методу `Sum2`. Тут же нужно сказать, что такой способ имеет два ограничения: у метода может быть только один параметр с модификатором `params`, и он должен быть последним, иначе среда не сможет отделить ~~мех от котлет~~ простые параметры от `params`. Это значит, что следующий метод корректен:

```
int Sum2(string str, params int[] values)
```

А вот эти методы уже являются ошибочными:

```
// params должен быть последним
int Sum2(params int[] values, string str);

// и может быть только один параметр params
int Sum2(params string[] str, params int[] values);
```

ПРИМЕЧАНИЕ

Исходный код этого примера можно найти в папке *Source\Chapter3\Parameterskinds* сопровождающего книгу электронного архива (см. *приложение*).

У параметров могут быть значения по умолчанию. Посмотрим на следующий метод:

```
public void IncSize(int inc = 1)
{
    ExpandSize(inc, inc, inc);
}
```

Метод принимает один параметр, после имени которого стоит знак равенства и значение по умолчанию. Если мы вызовем метод и укажем конкретное значение, то код будет работать так же, как и раньше:

```
Shed shed = new Shed();
shed.IncSize(10);
```

А если ничего не передать:

```
shed.IncSize();
```

то при вызове метода в параметре будет использоваться значение по умолчанию, в качестве которого мы указали единицу, т. е. размер сарая увеличится на 1.

3.3.3. Перегрузка методов

Допустим, что нам нужно создать еще один метод, который будет изменять только глубину и высоту сарая, без изменения его высоты. Как поступить в этом случае? Можно создать метод с новым именем типа `ChangeWidthAndLengthwiseOnly()`, но может возникнуть еще 10 ситуаций, когда понадобится создать схожий по функциональности метод, и каждый раз придумывать новые имена будет ужасным способом решения этой проблемы.

Проблема решается намного проще — мы можем создать метод `ExpandSize()`, который станет получать в качестве параметров только ширину и глубину:

```
public void ExpandSize(int x, int y)
{
    Width += x;
    Lengthwise += y;
}
```

Но у нас уже был написан такой метод в *разд. 3.3.2*, и не будет ли это ошибкой? Короткий ответ — нет. А как система узнает, какой метод нужно вызывать? По типам и количеству параметров и только по ним, а имя не имеет значения. Однако если вы попытаетесь после этого создать еще один метод для изменения только ширины и высоты, то вот тогда уже произойдет ошибка:

```
public void ExpandSize(int x, int h)
{
    Width += x;
    Height += h;
}
```

Несмотря на то что имена параметров здесь отличаются от предыдущего варианта, в котором мы тоже изменяли глубину и ширину, система воспримет оба эти варианта как одинаковые, потому что совпадает как количество параметров, так и их типы данных.

Использование нескольких методов с одним и тем же именем, но с разными параметрами — очень удобная возможность, которая называется *перегрузкой методов*. Мы будем использовать ее достаточно часто, особенно в конструкторах, к рассмотрению которых мы и переходим.

3.3.4. Конструктор

Чаще всего классы строятся вокруг одного-трех основных свойств. По крайней мере, это хорошие классы, которые решают только одну задачу и не пытаются объять необъятное. Наш класс `Shed` строился вокруг трех свойств, определяющих ширину, глубину и высоту сарая. После создания экземпляра класса мы должны задать

значения этих свойств. А что если кто-либо забудет установить свойства и начнет вызывать методы? В этом случае начнутся ошибки, и программа поведет себя некорректно.

Тут могут быть два выхода:

- ❑ абсолютно в каждом методе производить проверку, чтобы все основные свойства имели корректные значения;
- ❑ при создании объекта код должен сам задавать значения свойств уже на этапе инициализации.

Второй способ лучше, проще и предпочтительнее. Если у вас 10 методов и вы решили добавить в класс одно свойство, то по первому способу придется переделать и переписать все методы, чтобы они проверяли это свойство.

Итак, нужно задать значения на этапе инициализации, чтобы данные не нарушились и не привели к сбою, и тогда достаточно будет только контролировать устанавливаемые в свойства значения. Но как отловить момент инициализации? Для этого существуют специальные методы, называемые *конструкторами*. Конструктор — это метод, имя которого совпадает с именем класса, и он ничего не возвращает. Даже ключевое слово `void` не нужно писать. Например, следующий метод будет являться конструктором для сарая:

```
public Shed(int w, int l, int height)
{
    width = w;
    Lengthwise = l;
    this.height = height;
}
```

У конструктора тоже может быть модификатор доступа. Обычно мы будем создавать публичные конструкторы, потому что таким образом можно инициализировать класс и вызывать эти конструкторы. Приватный конструктор вызвать будет нельзя, и в этом есть смысл, но и используется такой конструктор очень редко.

В качестве параметров конструктор получает три переменные, с помощью которых уже на этапе создания объекта мы можем задать начальные значения для свойств объекта.

Я специально написал инициализацию и имена переменных так, чтобы мы рассмотрели различные варианты грабель, на которые можно наступить.

- ❑ Первая строка самая простая — нужно проинициализировать переменную `width`, и ей мы просто присваиваем значение. Если переменная должна защищаться, и ее свойство является не просто оберткой, а при изменении значения производится проверка, то лучше присваивать значение не переменной напрямую, а свойству, чтобы проверки отрабатывали. Наверное, это даже более предпочтительный вариант, и лучше использовать его.
- ❑ Во второй строке мы должны изменить свойство, для которого нет переменной, потому что мы объявляли его в сокращенном варианте. Тут уже деваться некуда, поэтому присваиваем значение именно свойству.

- В третьей строке у нас переменная, через которую передается значение, имеет точно такое же имя, что и переменная класса. Если мы напишем просто `height = height`, то как компилятор узнает, что нужно присвоить значение переданного параметра переменной, которая принадлежит классу? Тут нас спасает ключевое слово `this`. Что это за загадочное `this`? Это ключевое слово, которое всегда указывает на текущий объект, т. е. в нашем случае под словом `this` кроется объект класса `Shed`.

Получается, что запись `this.height` идентична записи `Shed.height`. А почему нельзя просто написать `Shed.height` без всяких `this`? Нельзя, потому что `Shed` — это класс, а переменная может принадлежать только объекту, если она не статична. В классе переменная лишь описывается, а создается она на этапе инициализации объекта. Но когда мы проектируем класс, то можем сослаться на будущий объект через ключевое слово `this`.

Теперь инициализация сарая будет выглядеть следующим образом:

```
Shed sh = new Shed(1, 2, 3);
```

Вот и все. Этот код создаст новый объект класса `Shed`, и наш новый сарай будет иметь размеры $1 \times 2 \times 3$. Теперь понятно, для чего скобки нужны при инициализации? Раньше они были пустыми, потому что вызывался конструктор по умолчанию. Да, конструктор существует всегда, и если вы не написали свой, то будет использоваться конструктор по умолчанию. Откуда он берется? Забегу вперед и скажу, что он наследуется от класса `Object`, от которого наследуются все классы в `C#`. Когда мы будем изучать наследование, вы увидите этот процесс на практике.

Несмотря на то что мы определили свой конструктор, конструктор по умолчанию никуда не пропадает за счет возможности перегрузки методов. Вы можете объявить одновременно два конструктора:

```
public Shed()  
{  
    ...  
}  
public Shed(int w, int l, int height)  
{  
    ...  
}
```

Поскольку конструктор — почти как метод, вы можете создать множество разных вариантов на все случаи жизни. Поэтому у программиста, который будет использовать наш класс, все еще остается возможность вызвать вариант по умолчанию, создающий сарай без инициализации значений.

Чтобы конструктор по умолчанию тоже инициализировал значения, мы должны написать собственный вариант конструктора без параметров, который как бы перекроет функциональность наследуемого варианта. Это можно сделать так:

```
public Shed()
{
    width = 1;
    Lengthwise = 1;
    this.height = 1;
}
```

Теперь, если вы создадите конструктор следующей строкой кода, то все размеры нового сарая будут равны единице, т. е. по умолчанию мы получим кубический сарай с длиной сторон, равной 1:

```
Shed sh = new Shed();
```

Хотя такой конструктор вполне допустим, я не рекомендую вам программировать таким способом. Почему? Допустим, у вас есть 10 конструкторов на все случаи жизни, и в каждом из них инициализируются параметры по-разному. Теперь возникла необходимость добавить в класс новое свойство — имя сарая. Как можно задать значение по умолчанию для него? Да очень просто, скажете вы, просто объявим свойство так:

```
string name = ""
public string Name
{
    get { return name; }
    set { name = value; }
}
```

Отличное решение! Действительно, мы можем инициализировать переменные класса во время их объявления, и я не зря привел этот пример. Если перед нами простая переменная, то при необходимости инициализируйте ее сразу же при объявлении значением по умолчанию, а не в конструкторах. Если потом понадобится, в конструкторе можно будет и переопределить (назначить другое) значение.

Еще одно интересное поведение .NET, которое нужно учитывать. Если в вашем классе вообще нет конструктора, то .NET будет использовать конструктор по умолчанию, который сделает инициализацию за вас, и он будет без параметров:

```
public class Shed() {
}
```

У этого класса нет ничего, но его объект можно создать, если вызывать конструктор без параметров:

```
Shed sh = new Shed();
```

А если у вас есть конструктор с параметрами?

```
public class Shed() {
    public Shed(int w, int h) {
        ...
    }
}
```

Да, тут у нас есть конструктор с параметрами, но мы не объявили варианта без параметров. И вот здесь уже строка:

```
Shed sh = new Shed();
```

не будет скомпилирована. При наличии хотя бы одного конструктора вариант по умолчанию из .NET уже использоваться не может. Вам придется явно в коде добавить перегруженную версию без параметров.

Усложняя задачу: допустим, нужно загрузить обои для стен в виде картинки. Обои — это уже не простой тип данных, картинки в C# — это отдельные специальные классы, тут может понадобится куча дополнительных действий. В частности, в таких случаях очень часто требуется выполнить несколько действий, которые объединяются в отдельный метод — например: `LoadTexture()`. Если его нужно вызвать на этапе инициализации, то придется добавлять его в каждый конструктор вашего класса. Может быть, есть способ лучше?

Способ по имени «лучше», конечно же, есть. В случае с нашим сараем конструктор по умолчанию можно написать так:

```
public Shed(): this(1, 1, 1)
{
    // здесь может быть еще код
}
```

Самое интересное кроется в записи после имени конструктора. Там стоят двоеточие и `this`, которому передаются три параметра. Вспоминаем, что такое `this`? Это же ключевое слово, которое всегда является текущим классом. Раз мы сараю передаем три параметра, не может ли это значить, что `this(1,1,1)` вызовет конструктор, который мы написали самым первым с тремя параметрами? Так и есть! Сначала будет вызван конструктор, который соответствует по параметрам, а потом будет выполнен код текущего конструктора.

В этом случае мы пошли от сложного к простому: написали самый сложный конструктор, а потом вызывали его из простого, просто недостающие параметры установили в единицу. Это не обязательно. Можно пройти и в обратном направлении, и обратное направление выгодно там, где нужно производить инициализацию вызовом методов. Например:

```
public Shed()
{
    LoadTexture();
}

public Shed(int w, int l, int height): this()
{
    // здесь может быть еще код
}
```

В этом случае основная инициализация идет в конструкторе по умолчанию, а конструктор с тремя параметрами просто вызывает его, освобождая вас от необходи-

мости писать еще раз код загрузки текстуры. Если простые переменные инициализировать во время объявления, то такой код будет тоже очень хорошим, ведь ширина, высота и глубина окажутся уже заданы.

Есть еще один вариант: написать метод с именем типа `InitVariables()`, где будут инициализироваться данные, общие для всех конструкторов. Теперь достаточно вызывать этот метод из всех конструкторов, и будет вам счастье.

3.3.5. Статичность

В английском языке тема, которую мы будем сейчас рассматривать, называется «static». В русских переводах можно встретить два варианта: «статичный» и «статический». Я приверженец именно первого перевода, потому что статичность — это неизменное состояние, и это как раз хорошо отражает суть проблемы. «Статический» в некоторых словарях тоже ассоциируют с неизменностью состояния, но меня почему-то это понятие чаще заставляет вспомнить про электричество.

Итак: *статичность* и ключевое слово `static`. Это ключевое слово относится не только к методам, но и к переменным, и начнем мы рассмотрение с первых.

Когда мы хотим получить доступ к методу класса, то должны обязательно создать экземпляр этого класса, т. е. создать объект. Класс не обладает памятью и не может что-то делать, потому что это всего лишь проект. Но как же тогда при запуске программы вызывается метод `Main()`? Разве это происходит только благодаря его магическому имени? По магическому имени система всего лишь находит, какой метод нужно вызвать, но то, что его можно вызвать без создания класса, — заслуга как раз ключевого слова `static`.

Итак, вы можете обращаться к статичным методам без создания класса! Но тут же возникает и ограничение: статичный метод может использовать только переменные, объявленные внутри этого метода (они могут быть любыми), или внешние по отношению к методу, но они должны быть обязательно статичными. К нестатичным внешним данным такой метод обращаться не может, потому что объект не создавался, а если кто-то и создавал объект, то статичный метод и данные к нему не относятся, поэтому непроинициализированные данные не могут быть доступны к использованию. В последнем утверждении кроется очень интересная особенность, которую стоит рассмотреть глубже.

Статичные методы и переменные создаются системой автоматически и прикрепляются к классу, а не к объекту. Да, именно к классу, т. е. к проекту, и при инициализации нового объекта память для статичных переменных не выделяется. Сколько бы объектов вы ни создавали из класса, всегда будет существовать только одна версия статичной переменной, и все как бы будут разделять ее.

Это очень интересное свойство статики, и классическим примером ее использования является возможность подсчета количества объектов, созданных из одного класса.

Давайте в нашем классе `Shed` создадим статичную переменную `ObjectNumber`, по умолчанию равную нулю. В конструкторе класса значение переменной станет уве-

личиваться на единицу, а открытый метод `GetObjectNumber()` будет возвращать значение переменной:

```
static int ObjectNumber = 0;

public int GetObjectNumber()
{
    return ObjectNumber;
}

public Shed()
{
    ObjectNumber++;
}
```

Теперь вы можете попробовать в методе `Main()` создать несколько экземпляров класса:

```
Shed shed = new Shed();
Console.WriteLine(shed.GetObjectNumber());
Shed shed1 = new Shed();
Console.WriteLine(shed1.GetObjectNumber());
```

Запустите класс и убедитесь, что после создания первого экземпляра значение переменной `ObjectNumber` стало равно 1, а после создания второго класса она не обнулилась, и у второго экземпляра класса значение переменной стало равно 2. Если создать еще один экземпляр, то переменная увеличится еще на единицу.

Нестатические переменные у каждого объекта свои, и изменение нестатического поля у одного экземпляра не влияет на другие экземпляры. Статическое поле всегда одно для всех, и оно разделяется между ними.

А как инициализировать статические переменные? Можно это сделать в простом конструкторе, но тогда переменная будет сбрасываться при создании любого экземпляра класса. Можно сделать в конструкторе какую-то проверку: если статическая переменная равна нулю, то инициализировать, иначе не трогать. Но это тоже не очень хороший выход.

Самый лучший способ — использовать статичный конструктор, который, как и простой конструктор, имеет такое же имя, что и класс, но объявлен с ключевым словом `static`:

```
static Shed()
{
    ObjectNumber++;
}
```

Такой конструктор обладает следующими свойствами:

- выполняется только один раз, вне зависимости от количества объектов, созданных из класса;

- ❑ не может иметь параметров, а значит, его нельзя перегружать, не получится создать более одного конструктора, и он будет выглядеть только так, как описано ранее;
- ❑ конструктор не имеет модификаторов доступа, потому что его не вызывают извне, он вызывается автоматически при создании первого объекта из класса или при первом обращении к статичному члену класса.

Почему нельзя перегружать конструктор и создать версию с параметрами? В третьем пункте я уже почти ответил на этот вопрос. Просто мы никогда не вызываем статичные конструкторы, они вызываются автоматически. Как только вы обращаетесь к какому-то статичному свойству класса, платформа автоматически вызовет статичный конструктор, и она не может знать, какие параметры вы хотите передать, поэтому вызывает его только без параметров.

Вы можете создавать даже целые статичные классы. Если класс объявлен как статичный, то он может содержать только статичные переменные и методы:

```
static class MyStaticParams
{
    public static void SomeMethod()
    {
    }
}
```

Тут нужно заметить, что такой класс не имеет смысла инициализировать для создания объекта, потому что все его члены доступны и без инициализации.

Если у вас есть класс, у которого все методы статичные и нет других данных, то при компиляции приложения будет получено предупреждение, что рекомендуется пометить класс как статичный. Это не ошибка, и если вы этого не сделаете, то программа все же будет выполняться, однако рекомендация эта сама по себе очень хорошая. Так что если у класса нет не статичных методов и свойств, то класс помечаем как статичный.

В статичные классы объединяют в основном какие-то вспомогательные переменные или методы, которые должны быть доступны для всего приложения. Раньше, когда языки не были полностью объектными, мы могли создавать переменные или методы, которые не принадлежали определенному классу. Такие переменные назывались *глобальными* и были видны во всем приложении. В чисто ООП-языке не может быть методов и переменных вне класса.

Статичные члены класса принадлежат классу, а не объекту, поэтому обращаться к ним нужно через имя класса. Нестатичные данные относятся к объекту, поэтому мы должны инициализировать объект из класса и обращаться к членам объекта через переменную объекта. Чтобы вызвать статичный член класса, нужно писать:

Имя_Класса.Переменная

или

Имя_Класса.Метод()

Например, статичный метод, который мы описали ранее, вызывается следующим образом:

```
MyStaticParams.SomeMethod();
```

Слева от точки здесь стоит имя класса, а не переменная объекта.

И последнее замечание относительно статичных переменных. Их значение инициализируется при первом обращении. В этот момент вызывается конструктор или выделяется память. То есть если вы в программе объявили 100 статичных переменных, это не значит, что сразу при старте программы все они будут проинициализированы и для всех будет выделена память. Если бы это было так, то запуск программы был бы слишком долгим и съел бы слишком много памяти уже на старте. На самом деле память выделяется по мере надобности. Как только вы обращаетесь к статичной переменной в первый раз, для нее выделяется память и туда заносится ее значение.

Несмотря на то что статичные методы и переменные вполне экономичны и иногда удобны с точки зрения использования, применяйте их только тогда, когда это реально необходимо.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter3\StaticProp* сопровождающего книгу электронного архива (см. *приложение*).

3.3.6. Рекурсия

Рекурсивный вызов метода — это когда метод вызывает сам себя. Да, такое действительно возможно. Вы без проблем можете написать что-то в стиле:

```
void MethodName()  
{  
    MethodName();  
}
```

Такой метод вполне корректен с точки зрения программирования и языка C#, но некорректен с точки зрения выполнения. Дело в том, что если вы вызовете метод *MethodName()*, то он будет бесконечно вызывать сам себя, и программа заикнется. Из этого бесконечного вызова метода нет выхода, и поэтому рекурсивные методы опасны с точки зрения программирования. Вам следует быть осторожным и убедиться, что из рекурсивного вызова обязательно будет выход, т. е. наступит такое состояние, при котором рекурсия прервется.

Очень часто можно встретиться с тем, что рекурсии изучают на примере факториала. Я сам это делал, потому что такой пример весьма нагляден. Но однажды я прочитал статью, в которой автор утверждал, что из-за нас — авторов учебных пособий — программисты начинают думать, что факториал действительно нужно вычислять через рекурсивный вызов. Если кто-то из прочитавших мои книги, где я учил рекурсии через факториал, тоже пришел к такому выводу, то могу извиниться и пообещать, что больше так учить не стану. Я это делал лишь из-за наглядности,

но не потому, что так нужно. Да, факториал можно вычислять рекурсией, но намного лучше, быстрее и эффективнее делать это через простой цикл, и в этой книге мы уже познакомились с факториалом во время изучения циклов, так что здесь придется придумать другой пример.

Немного подумав, я решил показать реальный пример рекурсии на таком примере, где виден результат, и выбрал для этого алгоритм быстрой сортировки. Алгоритм основан на том, что он делит массив по определенному признаку и для каждой части массива снова вызывает сортировку. На втором шаге метод получает уже меньший по размеру массив, который снова режется на две части. Таким образом, на каждом этапе массив делится на все более мелкие части.

Давайте посмотрим, как это делается в коде (листинг 3.2).

Листинг 3.2. Быстрая сортировка

```
class Program
{
    static int[] array = {10, 98, 78, 4, 54, 25, 41, 30, 87, 60, 84, 6, 12};

    static void Main(string[] args)
    {
        sort(0, array.Length-1);
        foreach (int i in array)
            Console.WriteLine(i);
        Console.ReadLine();
    }

    static void sort(int left, int right)
    {
        int i = left;
        int j = right;
        int x = array[(left + right) / 2];

        do
        {
            while (array[i] < x) i++;
            while (array[j] > x) j--;
            if (i <= j)
            {
                int y = array[i];
                array[i] = array[j];
                array[j] = y;
                i++;
                j--;
            }
        } while (i < j);
    }
}
```

```
foreach (int k in array)
    Console.Write(k.ToString() + ',');
Console.WriteLine(" ");

if (left < j)
    sort(left, j);

if (left < right)
    sort(i, right);
}
}
```

Разберемся со смыслом этой быстрой сортировки. У нас есть массив из чисел. Мы берем на удачу элемент посередине и запускаем цикл, в котором нужно сделать так, чтобы все элементы слева от середины списка были меньше этого значения, а справа от середины были больше этого значения. Для массива:

10, 98, 78, 4, 54, 25, **41**, 30, 87, 60, 84, 6, 12

средним значением будет 41, а значит, числа 98, 78 и 54 нужно перенести правее 41, а числа 30, 6 и 12 — левее, просто меняя эти значения местами. В результате получаем:

4, 6, 12, 10, 30, 25, **41**, 54, 87, 60, 84, 78, 98

Да, массив еще не отсортирован, но мы точно знаем, что слева от 41 числа меньше его, а справа — больше. Теперь запустим сортировку для левой части:

4, 6, 12, 10, 30, 25

и для правой:

54, 87, 60, 84, 78, 98

Для каждой из половин метод `Sort()` снова повторяет процесс: для массива 4, 6, 12, 10, 30, 25 мы берем среднее значение 12 и перемещаем все числа меньше его налево, а все, что больше, — направо: 4, 6, 10, **12**, 30, 25. Массив практически отсортирован, и мы снова вызываем сортировку для двух половин этого массива.

Все, что представлено в этом листинге, нам уже известно. Попробуйте сами пробежаться мысленно по коду и понять, как он выполняется. Рекурсивность в коде находится в самом конце, где метод `sort()` вызывает сам себя, передавая индексы массива, начиная с которого и по какой нужно просмотреть массив и при необходимости отсортировать.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter3\QuickSort* сопровождающего книгу электронного архива (см. приложение).

3.3.7. Деструктор

В разд. 3.3.4 мы обсуждали существование специализированного метода по имени *конструктор*. Это метод, который вызывается автоматически при создании объекта и призван выделять ресурсы и инициализировать переменные. А есть ли такой же метод, вызывающийся автоматически при уничтожении объекта, в котором можно было бы подчищать и освобождать выделенные ресурсы? Такой метод есть, но смысл его использования весьма расплывчат и не явен. Давайте сначала поговорим о том, почему этот так.

Платформа .NET достаточно интеллектуальна, чтобы самостоятельно освобождать все выделенные ресурсы. Это очень важно, потому что программисту не приходится думать о том, какие ресурсы и когда следует отпустить и уничтожить. Все за нас делает система, поэтому и не происходит утечек памяти.

Система ведет счетчики, в которых подсчитывается количество ссылок на экземпляры классов. Когда количество рабочих ссылок на объект становится равным нулю и объект больше не используется, он добавляется в список объектов, подлежащих уничтожению. По мере возможности и надобности запускается специальный модуль — *сборщик мусора*, который освобождает выделенные ресурсы автоматически. Тут есть один очень важный момент, который вы должны помнить и понимать: объекты не обязательно уничтожаются сразу же после того, как они становятся ненужными.

Вы можете самостоятельно принудить сборщик мусора к работе, чтобы он пробежался по классам, которые нуждаются в очистке, написав следующую строку кода в своей программе:

```
GC.Collect();
```

В .NET есть такой класс: `GC`, который позволяет работать со сборщиком мусора и вызывать его методы. Метод `Collect()` — это статичный метод, который инициирует сборку мусора.

Если вы пишете только для платформы .NET и только с помощью методов этой платформы, то использование деструктора не нужно и не имеет смысла, — платформа сама все подчистит. Деструктор может понадобиться в том случае, когда вы обращаетесь напрямую к функциям платформы Windows, не имеющей сборщика мусора. В ней нужно будет самостоятельно освободить любую память, которую вы выделили сами, вызывая методы Windows.

Метод, который вызывается автоматически в ответ на уничтожение объекта, называется *деструктором* и описывается точно так же, как в языке C++:

```
~Form1()  
{  
}  
}
```

На то, что это деструктор, указывает символ `~` (тильда) перед именем, а имя должно быть точно таким же, как и имя класса.

Это сокращенный вариант деструктора, но при компиляции сокращенный вариант превращается в полный вариант описания, который в .NET выглядит как метод `Finalize()`.

Если вы разрабатываете класс, который, например, открывает файл, то этот файл обязательно нужно закрыть после использования. Такое закрытие можно прописать в деструкторе, но лучше, помимо деструктора, реализовать еще метод с именем `Close()`. А что если пользователь забудет вызвать метод `Close()`? Если явно не закрывать файл, то он будет отмечен в системе как используемый, пока сборщик мусора не уничтожит объект. Это плохо — лучше использовать деструктор и в нем самостоятельно закрывать файл и освобождать его.

У деструктора нет параметров, как и у статичных конструкторов, потому что их выполняет система. Проблема деструктора в том, что он выполняется во время работы сборщика мусора, и если код деструктора будет медленным, то сборщик мусора будет тормозить и затормозит всю систему.

Сборщик мусор может передвигать объекты в памяти, и чтобы это сделать безопасно, на время сборки выполнение программы останавливается, так что если деструкторы будут медленные, то остановка приложения может быть значительной.

Если при уничтожении объекта нужно выполнить долгую операцию, то лучше воспользоваться методом: `Dispose()`. Это еще один способ выполнить код, который должен вызываться при завершении работы с объектом:

```
protected override void Dispose(bool disposing)
{
    if (disposing)
    {
        // здесь уничтожаем объекты
    }
    base.Dispose(disposing);
}
```

Пока нам нужно знать, что такой метод будет автоматически вызываться, если класс реализовал интерфейс `IDisposable`. Что такое интерфейс, мы еще узнаем в главе 6, а пока ограничимся только именем метода, потому что он реализован уже во множестве классов .NET платформы.

3.3.8. Упрощенный синтаксис

Если метод состоит только из одной строки, то его можно записать в упрощенном виде:

```
public int GetSize() =>
    Width * Height * Lengthwise;
```

Как можно видеть, здесь нет фигурных скобок, а вместо них после имени метода стоит символ `=>` и сразу же идет единственная строка метода.

3.4. Метод *Main()*

Метод с именем `Main` является самым главным методом в программе, потому что с него начинается выполнение приложения. Существует несколько вариантов написания метода `Main()`:

```
static void Main()
{
}

static int Main()
{
    return Целое_число;
}

static void Main(string[] args)
{
}

static public int Main(string[] args)
{
    return Целое_число;
}
```

Обратите внимание, что все варианты метода `Main()` являются статичными (в начале стоит модификатор `static`). Это значит, что метод можно вызывать без создания экземпляра класса. Логично? Я думаю, что да, ведь при запуске приложения еще никаких классов не создавалось, а значит, существуют только статичные методы и переменные, которые инициализируются автоматически при первом обращении.

Все эти объявления сводятся к одной простой истине: метод может возвращать пустое значение или число, а также может не принимать никаких параметров или принимать массив строк.

Метод `Main()` не обязан быть открытым и может быть объявлен как `private`. В этом случае другие сборки не смогут вызывать его напрямую. Если перед нами исполняемый файл, то он прекрасно будет запускаться и с закрытым методом `Main()`.

Почему в качестве параметра передается именно массив строк? Дело в том, что ОС, когда вызывает программу, может передать ей в качестве параметров одну строку. Это уже сама программа разбивает монолитную строку на массив, а в качестве разделителя использует пробел. Это значит, что если вы передадите программе два слова, разделенные пробелом, — например: `"parameter1 parameter2"`, то в массиве значений будут созданы две строки: `parameter1` и `parameter2`.

Следующий пример показывает, как отобразить в консоли все переданные параметры:

```
private static void Main(string[] args)
{
    foreach (string s in args)
```



```
    Console.WriteLine(s);  
    Console.ReadLine();  
}
```

Чтобы запустить пример из среды разработки и сразу же увидеть результат, вы можете прямо в среде разработки прописать параметры, которые должны будут передаваться программе. Для этого щелкните правой кнопкой мыши на имени проекта и в контекстном меню выберите пункт **Properties**. В нем найдите раздел **Debug** и в поле **Command line arguments** (Параметры командной строки) введите текст, который должен быть передан программе при запуске.

А что если нужно передать программе имя файла, которое содержит пробелы? Интересный вопрос, и мы часто можем встретиться с ним, но ответ на него находится легко: имя файла нужно заключить в двойные кавычки и передать в таком виде программе в качестве параметра. Точно так же можно поступить с любой другой строкой, которая не должна быть разбита по пробелам. Все, что находится между кавычками, не разбивается по параметрам.

Неужели доступ к параметрам командной строки можно получить только из метода `Main()`? А что если у нас большой проект, и нужно узнать, что нам передали из совершенно другого места? И это возможно. Существует такой класс: `Environment`, у которого есть статичный (это значит, что для доступа к методу не нужно создавать класс) метод `GetCommandLineArgs()`, который вернет нам массив аргументов. Следующий пример получает аргументы от класса и выводит их в консоль:

```
private static void Main()  
{  
    string[] args = Environment.GetCommandLineArgs();  
    foreach (string s in args)  
        Console.WriteLine(s);  
    Console.ReadLine();  
}
```

Фрагмент `Environment.GetCommandLineArgs()` возвращает массив параметров на один больше, потому что самым первым параметром (под индексом 0) всегда идет полный путь к запущенному файлу. В Интернете часто можно встретить вопрос о том, как узнать, откуда была запущена программа. Легко! Нужно посмотреть нулевой аргумент в `Environment.GetCommandLineArgs()`:

```
string fullpath = Environment.GetCommandLineArgs()[0];
```

Эта строка кода сохранит в переменной `fullpath` полный путь, включая имя файла запущенной программы.

ВНИМАНИЕ!

Метод `Main()` не может быть перегружен, т. е. в одном классе не может существовать несколько методов с этим именем, в отличие от любых других методов. Это связано с тем, что иначе ОС не сможет определить, какой из методов `Main()` является входной точкой программы.

3.5. Оператор *Using*

В *разд. 2.3* мы уже говорили о пространствах имен, но тогда некоторые вещи было еще рано объяснять, потому что они все равно были бы не понятны, поэтому сейчас я решил вернуться к этому вопросу и поговорить на эту тему более подробно. Вот тут нужно вспомнить первую строку исходного кода проекта `TestApplication` (см. *разд. 1.3.1*) и еще раз остановиться на ней:

```
using System;
```

Оператор `using` говорит о том, что мы хотим использовать пространство имен `System`. Теперь при вызове метода система сначала станет искать его у текущего объекта, и, если он не будет найден, произведет поиск в выбранном пространстве имен. Нужен пример? Он перед вами. Дело в том, что полный путь вызова метода `WriteLine()` выглядит следующим образом:

```
System.Console.WriteLine("Hello, World!!!");
```

У `System` есть класс `Console`, а у `Console` есть статичный метод `WriteLine()`. Чтобы не писать такую длинную цепочку, мы говорим, что находимся в пространстве `System`, и используем его методы и свойства.

Оператор `using` достаточно мощный, но не всесильный. Например, нельзя использовать пространство имен `System.Console`, и следующий код будет ошибочным:

```
using System.Console;

class EasyCSharp
{
    public static void Main()
    {
        WriteLine("Hello, World!!!");
    }
}
```

Ошибка произойдет из-за того, что `Console` является классом, а не пространством имен, и он не может быть выбран в качестве пространства имен. Единственное, что мы можем сделать, — создать псевдоним для класса, например:

```
using output = System.Console;

class EasyCSharp
{
    public static void Main()
    {
        output.WriteLine("Hello, World!!!");
    }
}
```

В этом примере с помощью директивы `using` мы создаем псевдоним с именем `output` для класса `Console` и используем этот псевдоним в коде проекта. Псевдонимы удобны в тех случаях, когда мы подключаем два различных пространства.

ПРИМЕЧАНИЕ

Исходный код этого примера можно найти в папке *Source\Chapter3\Alias* сопровождающего книгу электронного архива (см. приложение).

Пространства имен — это как папки для функций. В папке *System* лежит все, что касается системы, а в *System.Windows* можно найти все необходимое для работы с окнами. Таким образом, с помощью указания пространств имен вы говорите компилятору, с какими функциями и из какого пространства имен вы будете работать. Если не сделать этого, то придется вызывать функции, указывая их полный путь. Опять же, аналогия с файловой системой. Если войти в какой-либо каталог, то можно запускать имеющиеся в нем файлы, указывая только их имена. В противном случае приходится указывать полный путь.

Задавая с помощью *using* пространства имен, вы говорите компилятору, где искать функции, используемые в вашем коде. Но это не единственная задача, которую решает эта директива. В .NET очень много объектов, и даже в самой платформе есть некоторые объекты, которые имеют одинаковые имена. Например, есть кнопка для веб-формы, а есть кнопка для оконного приложения. Это совершенно разные кнопки, и они несовместимы. Чтобы компилятор узнал, какая именно из них нужна, вы и указываете пространство имен, с которым работаете. Например, если указано *System.Windows.Forms*, то, значит, вы используете оконное приложение, а если *System.Web.UI.Controls*, то перед нами веб-приложение.

А что если вам нужно в одном и том же модуле использовать и то и другое? Как указать, что *Button* — это оконная кнопка, а не веб? Конечно, можно указывать полный путь к объекту *System.Windows.Forms.Button*, но лучше создать псевдоним:

```
using winbutton = System.Windows.Forms.Button;
```

Таким вот методом мы как бы создали псевдоним для оконной кнопки с именем *winbutton* и можем обращаться к ней не как к *Button*, а как к *winbutton*. Точно так же можно создать псевдоним для веб-кнопки и обращаться к ней по короткому имени.

3.6. Объекты только для чтения

Простые переменные могут стать константами, и тогда их значения нельзя будет изменить во время выполнения программы. А можно ли сделать то же самое с объектом и написать что-то типа:

```
const Shed sh = new Shed();
```

Так делать нельзя. Дело в том, что значение константы должно быть известно уже на этапе компиляции и должно быть вполне конкретным. В рассматриваемом случае у нас переменная-объект. Какое значение компилятор должен будет поставить в программу вместо имени *sh*? Должно быть конкретное значение, а тут его нет.

Для чего вообще нужно создавать переменную объекта как константу? Чаще всего, чтобы защитить ее от повторной инициализации и выделения памяти. Если вам

нужно именно это, то объявите переменную как доступную только для чтения, а для этого перед переменной надо поставить ключевое слово `readonly`, как показано в следующем примере:

```
static readonly Shed sh = new Shed();

static void Main(string[] args)
{
    sh = new Shed(); // Ошибка, нельзя инициализировать повторно
}
```

3.7. Объектно-ориентированное программирование

Мы уже немного затронули тему объектно-ориентированного программирования (ООП) в *разд. 3.1* и выяснили, что такое *класс*. Сейчас нам предстоит познакомиться с ООП более подробно. Язык *C#* является полностью объектным, поэтому знание основных принципов этой технологии весьма обязательно для понимания материала книги и языка *C#*.

Основная задача ООП — упростить и ускорить разработку программ, и с такой задачей оно великолепно справляется. Когда я впервые познакомился с этой технологией в *C++*, то сначала не мог ее понять и продолжал использовать процедурное программирование, но когда понял, то ощутил всю мощь ООП и не представляю сейчас, как я жил раньше. Крупные проекты писать без классов невозможно.

Проблема при описании ООП заключается в том, что необходимо сначала получить слишком много теоретических знаний, прежде чем можно будет писать полноценные примеры. Конечно же, я мог бы сразу познакомить вас с созданием более сложных программ, но описание самого языка усложнилось бы. Поэтому мы взяли пока простой пример, но уже познакомились на практике с возможностью создания простых классов и методов.

ООП стоит на трех китах: инкапсуляция, наследование и полиморфизм. Давайте поймем и исследуем каждого кита в отдельности.

3.7.1. Наследование

Для описания *наследования* снова вернемся к строениям. Допустим, нам нужно создать объект, который будет описывать дом. Для этого необходимо наделить объект такими свойствами, как высота, ширина и т. д. Но вспомним пример из *разд. 3.1*, где мы описывали сарай. Он имеет те же параметры, только дом делается из другого материала и должен иметь окна, которые у сарая могут отсутствовать. Таким образом, чтобы описать дом, не обязательно начинать все с самого начала, — можно воспользоваться уже существующим классом сарая и расширить его до дома. Удобно? Конечно же, потому что позволяет использовать код многократно.

Следующий абстрактный пример показывает, как будет выглядеть создание объекта «Дом»:

```
Объект Дом происходит от Сарая
Начало описания
    Количество окон
Конец описания
```

В этом примере Дом происходит от Сарая. А это значит, что у дома будут все те же свойства, которые уже есть у сарая, плюс новые, которые добавлены в описании. Мы добавили к сараю свойство `Количество окон`, а свойства ширины, высоты и глубины будут наследованы от предка, и их описывать не нужно.

Когда мы объявили наследника, то, помимо свойств, получили и метод подсчета объема. Но дом может иметь более сложную, нежели сарай, форму, и не всегда его объем так просто подсчитать. Чтобы у объекта Дом была другая функция, достаточно в этом объекте объявить метод с тем же именем и написать свой код:

```
Объект Дом происходит от Сарая
Начало описания
    Количество окон

    Число ПолучитьОбъем()
Начало метода
    Подсчитать объем дома
Конец метода
Конец описания
```

Теперь объекты Дом и Сарай имеют метод с одним и тем же именем, но код его может быть для каждого объекта свой.

Наследование бывает достаточно сложным и может строиться по древовидной схеме — от одного класса могут наследоваться несколько других. Например, от сарая можно создать два класса: дом и будка. От дома можно создать еще один класс — многоэтажный дом. Таким образом, получится небольшое дерево, а у многоэтажного дома среди предков будут аж два класса: сарай и дом. Тут есть одна важная особенность: многоэтажный дом наследует два класса последовательно. Сначала дом наследует все свойства и методы сарая и добавляет свои методы и свойства. После этого многоэтажный дом наследует методы и свойства дома, которые уже включают в себя свойства и методы сарая.

Такое наследование называется *последовательным*, и оно вполне логично и линейно. Множественное (параллельное) наследование в C# невозможно. Это значит, что вы не можете создать класс многоэтажки, который будет наследоваться сразу же от дома и от подземной парковки. Придется выбрать что-то одно.

Перейдем теперь от теории к делу и на практике посмотрим, как происходит наследование.

Итак, введем какое-то базовое понятие для строений: `Building`. Как я уже говорил, каждый класс желательно держать в отдельном файле, поэтому добавим файл `Building.cs` к проекту и в него запишем следующий код:

```
class Building
{
    int width;
    int height;

    public int Width {
        get { return width; }
        set { width = value; }
    }

    public int Height {
        get { return height; }
        set { height = value; }
    }

    public int Lengthwise { get; set; }
}
```

Здесь мы подразумеваем, что все здания обладают размерами: шириной, высотой и длиной. А более сложные архитектурные строения будут состоять из таких вот простых строений.

У сарая тоже есть размеры: ширина, высота и длина, и чтобы не копировать данные, мы можем просто наследовать класс здания, т. е. наследовать его функциональность. Для этого при объявлении класса после имени ставим двоеточие и имя класса, который мы наследуем:

```
class Shed : Building
```

Из самого класса сарая теперь можно убрать объявления свойства `Width`, `Height` и `Height`, и класс сарая теперь будет выглядеть так:

```
class Shed : Building
{
    public Shed(): this(1, 1, 1)
    {
    }

    public Shed(int w, int l, int height)
    {
        Width = w;
        Lengthwise = l;
        this.Height = height;
    }

    public int GetSize()
    {
        int size = Width * Height * Lengthwise;
        return size;
    }
}
```

У нас здесь нет больше объявления свойств — мы их наследовали от предка `Building`. А раз наследовали, то имеем право использовать, — вот почему в коде все еще остались значения `Width`.

Класс `сарая` унаследовал свойства и добавил свои методы. У базового класса `Building` нет метода `GetSize`, а у `сарая` есть. Вообще-то метод `GetSize` можно было объявить в классе `Building`, и тогда этот метод был бы у них обоих, но ради примера я пока сделал именно так. Теперь посмотрим на пример использования:

```
Building myBuilding = new Building();
myBuilding.Height = 10;
myBuilding.Width = 20;
myBuilding.Lengthwise = 30;
// myBuilding.GetSize();

Shed myFirstShed = new Shed(10, 20, 30);
myFirstShed.Height = 15;
myFirstShed.GetSize();
```

Мы создаем объект класса `Building` и указываем его размеры, заполняя свойства, но строка получения размера закомментирована не просто так — ведь у этого класса нет такого метода, он есть лишь у `сарая`.

Потом мы создаем сарай и при его создании можем указать конструктору размеры, потому что такой специальный конструктор у `сарая` есть. Затем я меняю высоту `сарая` на 15 метров, и хотя это свойство отсутствует непосредственно у класса `Shed`, мы можем его менять, потому что это свойство есть у предка, от которого мы наследуем.

Давайте создадим еще один файл: `AppartmentBuilding.cs` — и в нем объявим класс для многоэтажного здания:

```
class AppartmentBuilding : Building
{
    public int Floors { get; set; }
}
```

Этот класс также наследует все возможности `Building` и добавляет одно новое свойство — количество этажей: `Floors`. Пример использования такого класса:

```
AppartmentBuilding appartmentBuilding = new AppartmentBuilding();
appartmentBuilding.Height = 10;
appartmentBuilding.Width = 20;
appartmentBuilding.Lengthwise = 30;
appartmentBuilding.Floors = 9;
```

У этого объекта мы можем задать не только размеры, но и количество этажей.

Теперь у нас есть базовый класс `Building`, который определяет размеры здания. Два класса наследуют этот класс и могут добавлять свои возможности — так, `AppartmentBuilding` добавляет свойство `Floors`, а `Shed` позволяет определить размер.

А если мы хотим, чтобы у многоэтажного дома была возможность определять размер здания, т. е. получить доступ к методу `GetSize()`? Можем мы наследовать сразу два класса?

```
class ApartmentBuilding : Building, Shed
```

Я попытался записать эти два класса через запятую, чтобы наследовать их оба, однако такой код завершится ошибкой, потому что наследовать можно только один класс. Класс может выступать базой для множества различных классов, но наследовать можно только один. Это как дерево, которое имеет только один ствол, на котором может расти много веток, на каждой ветке может расти много листьев, но один лист не может расти от нескольких веток (возможно, в природе такое бывает, но это уже аномалия).

Итак, как же решить такую задачу: получить доступ и к размеру, и к свойствам? У дерева ветка может расти из ветки, и то же самое в наследовании — мы можем наследовать класс, который уже наследует какой-то класс.

То есть если мы наследуем сарай, то унаследуем его свойство `GetSize`, а т. к. сарай наследует `Building`, то по цепочке мы получим доступ и к свойствам размера:

```
class ApartmentBuilding : Shed
```

Вот такой подход возможен — теперь у нас получилась цепочка:

```
Building < Shed < ApartmentBuilding
```

`Building` здесь задает базовую функциональность, `Shed` наследует свойства `Building` и добавляет свой метод, `ApartmentBuilding` наследует `Shed` и `Building` и добавляет свое свойство — количество этажей.

Таким способом можно строить достаточно сложные классы и наследования, но сразу предостерегу вас от него — большое количество классов в цепочке потом приводит к серьезным проблемам при поддержке такого кода. Используйте наследование, но аккуратно. Это вопрос уже хорошего тона в программировании и стоит отдельной книги.

3.7.2. Инкапсуляция

Инкапсуляция решает два вопроса. Во-первых, она определяет, что мы должны объединять данные и методы, которые работают над этими данными, в одну сущность. И во-вторых, дает возможность спрятать от конечного пользователя внутреннее устройство объекта и предоставить ему доступ только к тем методам, которые необходимы.

То есть за счет того, что мы объединяем данные и методы в один класс, мы предоставляем пользователю только те возможности, которые ему нужны для решения задачи, и прячем от него сложность устройства работы класса.

Посмотрим на следующий пример класса, который работает с файлом:

```
class File {  
    public int OpenHandle(string filename){  
    }  
}
```



```
public string ReadLine(int handle){  
    }  
}
```

Когда мы открываем файл `OpenHandle`, то после этой операции создается специальное значение, называемое `handle`, которое ассоциируется с открытым файлом. Именно его нам возвращает метод, и мы потом можем передать его методу `ReadLine`, чтобы прочитать строку из открытого файла. Казалось бы, это класс, и мы пишем объектный код, но на самом деле нет. Я просто тут объединил две процедуры, которые не сделали код объектным. Программисту надо иметь в виду, что есть некий `handle`, о котором нужно думать и который нужно куда-то передавать.

`Handle` — это данные, а метод, который читает данные из файла, — `read`. Согласно принципам инкапсуляции мы должны объединить методы и данные и спрятать сложность реализации:

```
class File {  
    private int handle  
    public File(string filename){  
    }  
    public string ReadLine(){  
    }  
}
```

Вот теперь это уже объектный код. У нас есть класс `File`, которому нужно передать имя:

```
File file = new File("readme.txt");
```

И у нас есть метод `ReadLine`, с помощью которого мы можем читать данные. Вся сложность реализации и все проблемы указателя на открытый файл скрыты от нас, потому что теперь это внутренние проблемы класса. Причем они не просто скрыты, но еще и защищены с помощью модификатора доступа `private`, чтобы пользователь не имел доступа к этим данным.

Свойство инкапсуляции «объединение методов и данных в класс» — это достаточно интересный вопрос, но не всегда его легко описать. Сложно дать четкие правила, какие данные объединять и с какими методами.

Второе свойство инкапсуляции четко связано с модификаторами доступа. По умолчанию в C# все методы закрыты. Если вы объявите метод в классе, то он сразу по умолчанию станет закрытым и к нему невозможно будет получить доступ, поэтому знание модификаторов и умение ими пользоваться является очень важным в ООП.

Что понимается под *пользователем объекта*? Существуют два типа пользователей: наследник и программист. Инкапсуляция позволяет прятать свою реализацию от обоих. Зачем программисту знать, как устроен объект, когда для его использования достаточно вызвать только один метод. Есть еще третий пользователь — сам объект, но он имеет доступ ко всем своим свойствам и методам.

Давайте вспомним следующую строку кода, которую мы уже рассматривали:

```
file.ReadLine()
```

Здесь происходит вызов метода `ReadLine()` класса `File`, который должен читать данные. Благодаря ООП и инкапсуляции нам абсолютно не нужно знать, как происходит чтение из файла. Достаточно знать имя метода, что он делает и какие параметры принимает, и вы можете использовать возможности метода, не задумываясь о его внутренностях и реализации.

Предоставление доступа осуществляется с помощью трех модификаторов доступа: `public`, `protected` и `private`, о которых мы уже говорили в *разд. 3.1*.

В следующем примере метод `OutString()` объявляется как закрытый (`private`):

```
class EasyCSharp
{
    public static void Main()
    {
        Shed shed = new Shed();
        Shed.OutString(); // Произойдет ошибка
    }
}
class Shed
{
    private void OutString()
    {
        Console.WriteLine("Hello World!!!");
    }
}
```

Поскольку метод объявлен закрытым, вы не можете вызвать `OutString()`, находясь в другом классе. При попытке скомпилировать проект компилятор сообщит о том, что он не видит метода, — ведь он закрытый.

Методы `private` доступны только текущему классу. Методы, которые объявлены как `protected`, могут быть доступны как текущему классу, так и его наследникам.

Как определить, какие методы должны быть открыты, а какие — нет? Для этого нужно четко представлять себе, к каким методам программист должен получать доступ, а к каким — нет. При этом программист не должен иметь доступа к тем составляющим объекта, которые могут нарушить целостность.

Объект — это механизм, который должен работать автономно и иметь инструменты, с помощью которых им можно управлять. Например, если представить автомобиль как объект, то водителю не нужно иметь доступ к двигателю или коробке передач, чтобы управлять автомобилем. Если у водителя будет к этим узлам прямой доступ, то он может нарушить работу автомобиля. Чтобы работа не была нарушена, водителю предоставляются специальные методы:

- ☐ педаль газа для управления оборотами двигателя;
- ☐ ручка переключения передач для управления коробкой передач;
- ☐ руль для управления колесами

и т. д.

Но есть компоненты, к которым водитель должен иметь доступ. Например, чтобы открыть дверь или багажник, не нужно выдумывать дополнительные механизмы, потому что тут нарушить работу автомобиля достаточно сложно.

Если ваши объекты будут автономными и смогут работать без дополнительных данных, то вы получаете выигрыш не только в удобстве программирования текущего проекта, но и в будущих проектах. Допустим, вы создали объект, который рисует на экране определенную фигуру в зависимости от заданных параметров. Если вам понадобятся подобные возможности в другом проекте, достаточно воспользоваться уже существующим кодом.

3.7.3. Полиморфизм

Полиморфизм в языках программирования и теории типов — способность кода обрабатывать данные разных типов.

Это одно из самых мощных средств ООП, которое позволяет использовать методы и свойства предков. Вспоминаем пример с объектами «сарай» (*shed*) и «дом» (*building*). У обоих этих объектов есть метод подсчета объема. Несмотря на то что методы разные (у дома мы переопределили этот метод), предок может обращаться к методу подсчета объема у потомка.

Вы можете объявить переменную типа класса предка, но присвоить ей значение класса потомка, и, несмотря на то, что переменная объявлена как предок, вы будете работать с потомком. Запутано? В C# все классы наследуются от *Object*. Это значит, что вы можете объявить переменную этого класса, а присвоить ей объект любого наследника (а т. к. это база для всех классов, то совершенно любого типа), даже сарай:

```
Object shed = new Shed();  
(Shed) shed.GetSize();
```

Несмотря на то что переменная *shed* объявлена как объект класса *Object*, мы присвоили ей объект класса *Shed*. В результате переменная останется сараем и не будет простым базовым объектом, а, значит, мы сможем вызывать методы сарая. Просто для этого нужно компилятору и исполняемой среде подсказать, что в переменной находится сарай, а для этого перед переменной в скобках нужно указать реальный тип переменной (класс *Shed*).

Поскольку *Object* — это базовый класс для всех классов, он дает нам возможность работать с любыми классами. Вспоминаем определение, которое я дал в начале этого раздела, — способность кода обрабатывать данные разных типов:

```
Object obj = new Shed();  
obj = new File();
```

Необязательно работать только с *Object*, можно работать с объектами через любой базовый класс, который является базовым для нашего.

Эту возможность желательно рассматривать на практике, поэтому мы пока ограничимся только определением, а дальнейшее изучение полиморфизма оставим на практические занятия.

В *разд 3.7.1* мы создали три класса `Building`, `Shed` и `AppartmentBuilding`. Так как первый является базовым для двух других, то следующие три строки являются в ООП абсолютно легальными благодаря полиморфизму:

```
Building building = new Building();
Building shed = new Shed();
Building apartment = new AppartmentBuilding();

Console.WriteLine("building" + building.ToString());
Console.WriteLine("shed" + shed.ToString());
Console.WriteLine("apartment" + apartment.ToString());
```

Здесь сначала объявлены три переменные, и все они объявлены как `Building`. Первой переменной присваивается результат создания объекта `Building`, второй переменной — объект `Shed`, третья будет представлена как `AppartmentBuilding`. Всё это легально и обеспечивается способностью кода обрабатывать данные разных типов. Нужно только, чтобы они были совместимы.

Последние три строки выводят на экран результат метода `ToString()`. Мы не создавали такой метод у строения, сарая или квартирного дома, но он есть, а откуда берется, мы узнаем в *разд. 3.8*. Сейчас же он выводит на экран тип метода, и мы увидим:

```
buildingTestApplication.Building
shedTestApplication.Shed
apartmentTestApplication.AppartmentBuilding
```

Несмотря на то что каждая из переменных объявлена как `Building`, при выводе видно, что они на самом деле разного типа.

А сможем ли мы получить размер сарая?

```
shed.GetSize();
```

Именно таким образом — нет, потому что переменная `shed` объявлена как `Buidling`. Да, в переменной реально находится сарай, но об этом знаем мы, а коду все равно, потому что он видит тип переменной `Building` и не проверяет, что на самом деле находится в этой переменной. Тут наша задача подсказать компилятору, что именно находится в переменной, но для этого нужно еще познакомиться с *приведением типов*.

Чуть ранее я отметил, что сохранение в переменных другого типа разрешено, нужно только чтобы они были совместимы. Что это значит?

В нашем примере все переменные объявлены зданиями, и в них мы сохраняем и сарай, и квартирный дом. Так как сарай наследует здание, то в сарае есть вся функциональность здания, т. е. типы совместимы.

А что если мы попробуем в квартирный дом поместить здание:

```
AppartmentBuilding apartment = new Building();
```

Такой код завершится ошибкой. У квартирного дома есть свойство `Floors`, которого нет у `Building`, а значит, совместимость нарушена, — в переменную можно помес-

тить объекты только тех типов данных, которые наследуют тип переменной. Если переменная строения `Building`, то в нее можно поместить лишь те классы, которые наследуют `Building`.

3.8. Наследование от класса *Object*

Все классы имеют в предках класс `Object`. Это значит, что если вы не напишете, от какого класса происходит ваш класс, то по умолчанию будет автоматически добавлен `Object`. Мы уже об этом говорили вскользь в *разд. 3.3.4*, а сейчас пришла пора обсудить эту тему основательно и заодно расширить знакомство с наследованием. Итак, два следующих объявления класса абсолютно идентичны:

```
class Person
{
}

class Person: System.Object
{
}
```

Во втором объявлении я специально указал полный путь к классу `Object`, включая пространство имен. На практике пространство имен писать не нужно, если оно у вас подключено к модулю с помощью оператора `using`.

За счет наследования любой класс в `C#` наследует следующие методы класса `Object`:

- ❑ `Equals()` — сравнивает переданный в качестве параметра объект с самим собой и возвращает `true`, если объекты одинаковые. По умолчанию сравнение происходит по ссылке, т. е. результатом будет `true`, когда переданный объект является тем же самым объектом, что и текущий. Вы можете переопределить этот метод, чтобы он сравнивал не ссылку, а состояние объекта, т. е. возвращал `true`, если все свойства объектов идентичны;
- ❑ `GetHashCode()` — возвращает хеш-значение текущего объекта в памяти;
- ❑ `GetType()` — возвращает объект класса `System.Type`, по которому можно идентифицировать тип объекта;
- ❑ `ToString()` — превращает класс в строку;
- ❑ `Finalize()` — автоматически вызывается, когда объект уничтожается;
- ❑ `MemberwiseClone()` — создает и возвращает точную копию объекта.

Вы можете переопределять эти методы, чтобы наделить их своей функциональностью. Давайте создадим новое консольное приложение и будем тренироваться на нем, как на кошках. Я назвал свой проект `PersonClass`. Сразу же добавьте к проекту еще один файл, где мы будем описывать наш собственный класс `Person`. Для этого щелкните правой кнопкой на имени проекта в панели **Solution Explorer**, из контекстного меню выберите **Add | New Item**, в окне выбора типа файла найдите **Class**

и укажите его имя: `Person`. Создав файл класса, добавим в него пару свойств и конструктор для удобства:

```
class Person
{
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Класс объявляет два свойства: `FirstName` и `LastName` — для хранения имени и фамилии соответственно. Конструктор класса получает в качестве параметров строковые переменные, которые сохраняются в свойствах имени и фамилии соответственно, инициализируя их начальными значениями.

Теперь попробуем вызвать метод `ToString()`, который был унаследован от класса `Object`:

```
Person p = new Person("Михаил", "Фленов");
Console.WriteLine(p.ToString());
```

В результате этого на экране вы должны увидеть строку: **PersonClass.Person**. В нашем случае `PersonClass` — имя пространства имен, в котором объявлен класс (я просто его опускал ранее, когда показывал код), так что метод `ToString()` по умолчанию показывает полное имя класса.

Каждый раз, когда вы захотите представить класс в виде строки, вместо чего-либо вразумительного вы будете видеть его полное имя. Это далеко не всегда удобно, поэтому метод `ToString()` переопределяется программистами чаще всего.

3.9. Переопределение методов

Двинемся дальше и посмотрим на примере, как можно переопределять методы. Для того чтобы метод можно было переопределять, он должен быть объявлен в базовом классе с ключевым словом `virtual`. В `C#` большинство открытых методов позволяют переопределение в наследниках, в том числе и метод `ToString()`. Мы уже знаем, что по умолчанию он выведет полное имя текущего класса, но давайте сделаем так, чтобы он выводил на экран имя и фамилию человека, данные которого хранятся в классе.

Метод `ToString()` объявлен в классе `Object` следующим образом:

```
public virtual string ToString()
```

Информацию о том, как объявлен метод, можно найти в MSDN. Сокращенный вариант объявления можно увидеть во всплывающей подсказке, если поставить

курсор на имя метода и нажать комбинацию клавиш <Ctrl>+<K>, <I>. Это значит, что в нашем классе `Person` мы можем переопределить метод следующим образом:

```
public override string ToString()
{
    return FirstName + " " + LastName;
}
```

Самое главное здесь — это имя, параметры и возвращаемое значение. В нашем случае имя — это `ToString`, параметров нет, возвращаемое значение — строка.

Теперь метод возвращает содержимое свойств имени и фамилии через пробел.

Магическое слово `override` в объявлении метода говорит о том, что мы переопределяем метод, который был у предка с таким же именем. Если не написать такого слова, то компилятор выдаст ошибку и сообщит, что у предка класса `Person` уже есть метод `ToString()`, и чтобы переопределить метод, нужно использовать ключевое слово `override` или `new`. У этих ключевых слов есть существенное различие: слово `override` говорит о том, что мы хотим полностью переписать метод, а слово `new` — что мы создаем свою, независимую от предка, версию того же метода.

Давайте увидим это на примере. Допустим, что у нас есть следующий код:

```
Person p = new Person("Михаил", "Фленов");
Console.WriteLine(p.ToString());

Object o = p;
Console.WriteLine(o.ToString());
```

Сначала мы создаем объект `Person` и вызываем его метод `ToString()`, а потом присваиваем объект `Person` объекту класса `Object` и выводим его метод `ToString()`. Так как `Object` — это предок для `Person`, то операция присвоения пройдет на ура.

При первом вызове `ToString()` логика работы программы понятна — будет вызван наш метод, который мы переопределили в `ToString()`. А что выведет `ToString()` во втором случае? — ведь мы вызываем его для класса `Object`, пусть и присвоили переменной класс `Person`... Вот тут уже все зависит от того, с каким ключевым словом мы переопределили метод:

- ❑ если это было `override`, то, как бы мы ни обращались к методу `ToString()` (напрямую или через предка), будет вызван наш переопределенный метод;
- ❑ если же мы используем ключевое слово `new`, то во втором случае будет выведено полное имя класса, т. е. будет вызван метод `ToString()` класса `Object`, несмотря на то, что в переменной находится объект класса `Person`. Ключевое слово `new` не перекрывает реализацию метода классов предков, и вы можете получить к ним доступ! И это очень важное различие.

Давайте переопределим еще один метод, который программисты переопределяют достаточно часто: `Equals()`. Этот метод должен возвращать `true`, если переданный в качестве параметра объект идентичен текущему. По умолчанию он сравнивает ссылки. Посмотрим на следующий пример:

```
Person p1 = new Person("Михаил", "Фленов");  
Person p2 = new Person("Михаил", "Фленов");  
Person p3 = p1;  
Console.WriteLine(p1.Equals(p2));  
Console.WriteLine(p1.Equals(p3));
```

Здесь объявлены три переменные класса `Person`. Первые две переменные имеют одинаковые значения свойств, но если их сравнить с помощью `Equals()`, то результатом будет `false`. Переменная `p3` создается простым присваиванием из переменной `p1`, и вот если сравнить их с помощью `Equals()`, то в этом случае мы уже получим `true`. Еще бы, ведь обе переменные являются одним и тем же объектом, потому что `p3` не инициализировалась — ей было просто присвоено значение `p1`.

Давайте сделаем так, чтобы и в первом случае результат сравнения тоже был `true`:

```
public new bool Equals(Object obj)  
{  
    Person person = (Person)obj;  
    return (FirstName == person.FirstName) &&  
        (LastName == person.LastName);  
}
```

В первой строке кода я присваиваю объект `obj` переменной класса `Person`. Это можно делать и не вызывает проблем, если в переменной мы действительно будем передавать переменную класса `Person`.

Далее ключевое слово `return` говорит, что нужно вернуть значение. А что оно вернет? Вот тут интересно, потому что возвращает оно результат сравнения двух свойств, связанных с помощью операции `&&`. Символы `&&` говорят о том, что результатом операции будет `true`, если сравнение слева и сравнение справа равны `true`. Если хотя бы одно из сравнений равно `false`, то результатом будет `false`. Получается, что если оба свойства у объектов совпадают, то можно говорить, что объекты одинаковые, и будет возвращено `true`. Теперь код сравнения, который мы писали раньше для переменных `p1`, `p2` и `p3`, во всех случаях вернет `true`.

Почему я здесь использовал именно слово `new`, а не `override`? Просто я знаю, что мне может понадобиться метод сравнения предка, и я не хочу его переопределять полностью. Тут есть интересный трюк, которым я хочу с вами поделиться.

Давайте напишем в классе `Program` статичную функцию, которая будет сравнивать два объекта класса `Person` и должна нам сообщать, являются ли они одинаковыми по параметру (это возвращает наш метод `Equals()`) или одинаковыми абсолютно, т. е. являются одним объектом (это возвращает `Equals()` по умолчанию). Если бы `Equals()` была объявлена как `override`, то такой финт не прошел бы, а в случае с `new` все решается легко:

```
static string ComparePersons(Person person1, Person person2)  
{  
    bool equalParams = person1.Equals(person2);
```



```
Object personobj = person1;  
bool fullEqual = personobj.Equals(person2);  
  
if (fullEqual)  
    return "Абсолютно одинаковые объекты";  
if (equalParams)  
    return "Одинаковые свойства объектов";  
  
return "Объекты разные";  
}
```

Сначала мы просто сравниваем два объекта, и будет вызван переопределенный метод. Так мы узнаем, являются ли объекты одинаковыми по параметру. После этого присваиваем первую переменную переменной класса `Object`, и теперь будет вызван метод класса `Object()`.

Слово `override` можно использовать только для методов, которые объявлены как виртуальные. У `Object` есть метод `GetType`, который не является виртуальным, и разработчики .NET не хотят, чтобы мы переопределяли этот метод. Если вы попытаетесь переопределить его:

```
public override System.Type GetType()  
{  
    return typeof(Person);  
}
```

то при попытке скомпилировать проект произойдет ошибка: **error CS0506: 'Person.GetType()': cannot override inherited member 'object.GetType()' because it is not marked virtual, abstract, or override** («ошибка CS0506: 'Person.GetType()': нельзя переопределить наследуемый метод 'object.GetType()', потому что он не помечен как виртуальный, абстрактный или перезаписанный»).

Абстрактные методы мы еще не проходили — это будет отдельная история. Виртуальные мы уже знаем, они могут переопределяться. Почему можно переопределять метод, который у предка помечен как `override`? Если предок переопределил метод с помощью `override`, значит, есть предок предка, который помечен как `virtual`. Иначе `override` был бы невозможен.

И хотя мы не можем переопределить метод, но можем создать новую версию в своем классе с указанием `new`. Она будет жить в нашем классе, а при обращении к методу через предка будет вызываться оригинальная версия, и это то, что нужно .NET:

```
public new System.Type GetType()  
{  
    return typeof(Person);  
}
```

Ключевое слово `new` указывать желательно, но не обязательно. Если просто написать:

```
public System.Type GetType()
{
    return typeof(Person);
}
```

код все еще скомпилируется, будет только выдано предупреждение: **warning CS0108: 'Person.GetType()' hides inherited member 'object.GetType()'. Use the new keyword if hiding was intended** («предупреждение CS0108: 'Person.GetType()' прячет унаследованный метод 'object.GetType()'. Используйте ключевое слово `new` если вы сделали это намеренно»).

Здесь нас предупреждают, что компилятор подразумевает, что мы прячем унаследованный метод, а не переопределяем его, но просит, чтобы мы указали `new`, чтобы подтвердить свои намерения спрятать метод и таким образом избавиться от предупреждения.

До сих пор мы переопределяли методы класса `Object`, но в своих классах вы также можете использовать виртуальные методы. Следующий класс объявляет метод `Work`, который объявлен как виртуальный. Это значит, что если будут созданы потомки от класса `Person`, то они смогут переопределить этот метод:

```
class Person
{
    public virtual void Work()
    {
        // здесь код метода
    }
}
```

Метод ничего не делает, потому что это неважно. Главное, что мы смогли определить метод:

```
class Driver: Person
{
    public override void Work()
    {
        // здесь код метода
    }
}
```

3.10. Обращение к предку из класса

А что если нам нужно написать метод в стиле `ComparePersons()`, но только внутри класса `Person`? Как внутри класса обращаться к предку? Тут есть очень хороший способ — ключевое слово `base`. Если `this` всегда указывает на объект текущего класса, то `base` указывает на предка. Вот как может выглядеть метод `ComparePersons()`, если его реализовать внутри класса `Person`:

```
class Person
{
    ...
    public string ComparePersons(Person person)
    {
        bool equalParams = Equals(person);
        bool fullEqual = base.Equals(person);
        if (fullEqual)
            return "Абсолютно одинаковые объекты";
        if (equalParams)
            return "Одинаковые свойства объектов";

        return "Объекты разные";
    }
}
```

В первой строке вызывается метод сравнения `Equals()`, который мы переопределили, а во второй строке с помощью ключевого слова `base` мы обращаемся к методу предка. Такой вариант выглядит красивее? На мой взгляд, да. Мало того, что сам код красивее, так еще и метод реализован внутри класса `Person`, где ему и место, чтобы сохранить логическую завершенность объекта.

У слова `base` есть одна интересная особенность — оно указывает на предка для текущего класса, и к нему нельзя обратиться, находясь в другом классе. И самое главное: ключевому слову `base` все равно, как мы переопределили метод. Это значит, что оно всегда вызовет метод `Equals()` предка, даже если мы переопределили его в своем классе с помощью `override`. Запомните эту интересную особенность.

Вернемся к классу `Person`, который мы уже использовали ранее. Напомню, как выглядят его объявление и конструктор:

```
class Person
{
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Здесь есть проблема: у конструктора имеются параметры. Если бы их не было, то мы могли бы создавать потомков без проблем. Но теперь, если создать потомка:

```
class Driver : Person
```

то в новом классе вы обязаны будете создать конструктор, вызывающий конструктор предка с такими же параметрами. Вспоминаем, что если у нас два конструктора, то из одного из них мы можем вызвать другой с помощью слова `this`. Вполне логично, что мы должны иметь возможность вызвать конструктор предка через `base` точно таким же способом. Так и есть:

```
class Driver : Person
{
    public Driver(string firstName, string lastName)
        : base(firstName, lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }
}
```

Если не вызвать конструктор базового класса, то произойдет ошибка компиляции.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter3\PersonClass* сопровождающего книгу электронного архива (см. *приложение*).

3.11. Вложенные классы

Все это время мы объявляли классы непосредственно внутри пространства имен. То есть создавали независимые классы. Но классы могут быть зависимыми, например, как показано в листинге 3.3.

Листинг 3.3. Вложенный класс

```
public class Shed
{
    // Здесь идут свойства и методы класса Shed

    public class Window
    {
        // Здесь идут свойства и методы класса Window

        public void ShutWindow()
        {
            // Код метода
        }
    }

    // объявляем переменную типа окна
    Window window = new Window();

    // превращаем переменную window в свойство
    public Window FrontWindow
    {
        get { return window; }
        set { window = value; }
    }
}
```

В этом примере мы объявили класс `сарая`, а внутри этого сарая объявлен другой класс `Window`, который будет реализовывать окно. Тут же в классе `Shed` я объявил переменную класса `Window`, написал код инициализации и даже превратил переменную в свойство, чтобы программист извне мог закрыть или открыть окно. Тут нужно заметить, что для того, чтобы переменную класса `Window` можно было превратить в свойство, класс должен быть объявлен как `public`, т. е. его спецификация должна быть открытой, иначе он не сможет быть свойством. Посмотрим теперь на пример использования:

```
Shed sh = new Shed();  
sh.Window.ShutWindow();
```

Здесь создается объект класса `sh` и вызывается метод `ShutWindow()` окна, которое находится внутри сарая.

Когда класс объявлен внутри другого класса, он называется *вложенным* (nested) внутрь другого класса. Создавайте вложенные классы, если это реально необходимо, — как здесь, где класс `Window` специфичен именно для этого сарая. Если вы захотите добавить такое же окно для другого класса — например, для машины, то придется писать класс заново, т. е. дублировать код. Так что если класс достаточно универсален, то лучше его объявить как независимый, чтобы его можно было использовать в других классах.

И вообще, если вкладывать классы друг в друга, то читаемость кода будет не очень хорошей, поэтому я в своей жизни создавал, может, пару вложенных классов, и в обоих случаях они были очень маленькими.

Пример из листинга 3.4 показывает, как сарай мог бы использовать независимый класс окна с тем же успехом.

Листинг 3.4. Решение задачи без использования вложенного класса

```
public class Window  
{  
    // Здесь идут свойства и методы класса Window  
    public void ShutWindow()  
    {  
        // Код метода  
    }  
}  
  
public class Shed  
{  
    // Здесь идут свойства и методы класса Shed  
  
    Window window = new Window();  
    public Window FrontWindow  
    {  
        get { return window; }  
        set { window = value; }  
    }  
}
```

Если вложенный класс объявлен как `private`, то его экземпляр может создать только объект такого же класса или объект, родительский для него. При этом объект не будет виден наследникам от родительского (любым наследникам от класса `Window`). Объекты класса `protected` могут создаваться как родительским объектом, так и наследниками от родительского.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter3\WestedClass` сопровождающего книгу электронного архива (см. приложение).

3.12. Область видимости

Нужно также понимать, что, помимо модификаторов доступа, есть и другие способы обеспечить видимость переменных внутри класса и функций. Посмотрите на следующий пример объявления класса (листинг 3.5).

Листинг 3.5. Область видимости переменной

```
using System;

class EasyCSharp
{
    int sum = 1, max = 5;

    public static void Main()
    {
        int i = 2;
        do
        {
            sum *= i;
            i++;
        } while (i <= max);
        Console.WriteLine(sum);
    }
}
```

Этот класс рассчитывает факториал с помощью цикла `do...while`, но это сейчас не имеет значения. Нас больше интересует время жизни переменных. Переменные `sum` и `max` объявлены внутри класса, но вне функции `Main()`.

Если объявление переменной находится вне метода, но внутри класса, то она доступна любому методу класса, независимо от используемых модификаторов доступа.

Если переменная объявлена внутри метода, то она доступна с момента объявления и до соответствующей закрывающей фигурной скобки. После этого переменная становится недоступной. Что значит «до соответствующей закрывающей фигурной

скобки»? Если переменная объявлена внутри метода, то она будет доступна до конца метода. Такой является переменная `i` в листинге 3.5.

Если переменная объявлена внутри цикла, то она видима только до конца этого цикла. Так, например, мы очень часто используем переменную `i` в качестве счетчика цикла:

```
for (int i = 0; i < 10; i++)  
{  
    // Код цикла  
}  
// здесь любой оператор за циклом
```

Как только завершится цикл, и управление программой уйдет за пределы закрывающей фигурной скобки цикла, т. е. на первый оператор за пределами цикла, переменные, объявленные внутри этого цикла, теряют свою актуальность и могут быть уничтожены сборщиком мусора.

Разрабатывая класс, я всегда устанавливаю его членам самый жесткий модификатор доступа `private`, при котором методы доступны только этому классу. Чтобы сделать член класса закрытым, вы можете не указывать перед методом или переменной модификатора доступа, потому что `private` используется по умолчанию.

В процессе программирования, если мне нужно получить доступ к методу класса родителя, из которого происходит мой класс, я в классе родителя перед нужным мне методом ставлю `protected`. Когда нужно получить доступ к закрытому методу моего класса извне, то только тогда я повышаю уровень доступа к методу до `public`. Если метод не понадобился извне, то он остается `private` по умолчанию.

Если возникает необходимость получить доступ к переменной, которая находится в другом классе, то никогда нельзя напрямую открывать доступ к переменной внешним классам. Самая максимальная привилегия, которую можно назначать переменной, — `protected`, чтобы наследники могли с ней работать, но `public` — никогда! Доступ к переменным может быть предоставлен только превращением ее в свойство или с помощью написания отдельных открытых методов внутри класса, что примерно идентично организации свойства.

Да, вы можете для закрытой переменной класса создать методы в стиле `SetParamValue()` и `GetParamValue()`, которые будут безопасно возвращать значение закрытой переменной и устанавливать его. Это не нарушает принципы ООП, но я все же предпочитаю организовывать свойство и использовать пару ключевых слов `get` и `set`.

Для защиты данных иногда бывает необходимо запретить наследование от определенного класса, чтобы сторонний программист не смог создать наследника и воспользоваться преимуществом наследования с целью нарушить работу программы или получить доступ к членам классов, которые могут нарушить ее работу. Чтобы запретить наследование, нужно поставить перед объявлением класса ключевое слово `sealed`:

```
// объявление завершеного класса
sealed class Person
{
    // члены класса
}

// следующее объявление класса приведет к ошибке
public class MyPerson : Person
{
}
```

В этом примере наследование от `Person` запрещено, поэтому попытка создать от него наследника приведет к ошибке.

3.13. Ссылочные и простые типы данных

Мне постоянно приходится возвращаться к темам, которые мы уже ранее обсуждали. Я стараюсь вести рассказ постепенно, вводить новые понятия по мере усвоения старых и не забегать вперед, поэтому разговор иногда прерывается на уточнение изученного материала.

Теперь, когда мы узнали про различные переменные и классы, я хочу ввести два понятия: ссылочный тип данных и значения. Как мы уже знаем, переменная — это как бы имя какой-то области памяти, в которой хранятся данные. Существуют два типа переменных: переменные-ссылки и переменные-значения.

Приложение использует два вида памяти: стек и кучу:

- ❑ *стек* — это область памяти, зарезервированная для программы, в которой программа хранит значения, которые должны быть поблизости. Он построен по принципу стопки тарелок. Вы можете положить очередную тарелку (переменную) сверху стопки и можете снять тарелку (уничтожить) тоже только сверху со стопки. Нельзя достать тарелку из середины или снизу, не снимая все вышележащие тарелки, но вы можете к ней прикоснуться (назначить переменной имя), дотронуться (прочитать значение) или нарисовать что-то маркером (изменить значение);
- ❑ *куча* — это большая область памяти, из которой вы можете запрашивать для программы фрагменты памяти большого размера и делать с ней что угодно, главное — не выходить за пределы запрошенной памяти, а за этим следит .NET. Если стек ограничен в размерах, то размер памяти кучи ограничен только размерами ресурсов компьютера. Именно ресурсов компьютера, а не оперативной памяти, потому что ОС может сбрасывать информацию на диск, освобождая память для хранения новой информации. Вы можете выделять память по мере надобности и освобождать ее, когда память не нужна.

Теперь посмотрим, где и как выделяется память для переменных. Для *простых типов* данных, таких как `int`, `bool`, `char`, размеры которых фиксированы, и система эти размеры знает, выделяется память в стеке. В ячейке памяти стека хранится непосредственно само значение.

Ссылочные типы создаются с помощью оператора `new`. В этот момент система выделяет память в куче для хранения объекта и выделяет память в стеке для хранения ссылки на область памяти в куче. Наша переменная представляет собой имя области памяти в стеке, и в этой области будет храниться только адрес области памяти в куче, где уже хранится сам объект.

Вот тут кроется очень интересная мысль, которую следует понять. Когда мы объявляем простую переменную, то для нее тут же готова память в стеке, и туда можно сохранить значение. Когда мы объявляем ссылочную переменную, в стеке выделяется память для хранения ссылки, но эта ссылка еще нулевая, и ее использование запрещено. Только когда ссылка будет проинициализирована с помощью `new` и в куче будет выделена необходимая память, мы сможем использовать ссылочную переменную. А, вот для чего мы вызываем `new`!

В приложениях Win32 программистам приходится самим заботиться о выделении и уничтожении выделяемой памяти. В .NET вы должны думать только о выделении, а платформа уже сама будет высвобождать память, когда она не используется. Но понимать разницу между ссылочными переменными и простыми все равно необходимо, чтобы вы лучше понимали, что происходит, когда используете оператор `new` и инициализируете объект.

3.14. Проверка класса объекта

Иногда бывает необходимо узнать, является ли объект экземпляром определенного класса. Такую проверку простым сравнением объекта и класса сделать невозможно. Для этого существует специализированное ключевое слово `is`:

```
Figure shape = new CircleFigure();  
if (shape is CircleFigure)  
    // Выполнить действие
```

В этом примере мы с помощью `is` проверяем, является ли объект `shape` классом `CircleFigure`. А он таковым является, несмотря на то, что переменная объявлена как `Figure`.

Тут же хочу показать вам еще одно ключевое слово: `as`, которое позволяет приводить типы классов. До этого, когда нужно было воспринимать один объект как объект другого класса, мы писали нужный класс в скобках. Например, в следующей строке кода я говорю, что объект `shape` надо воспринимать как `CircleFigure`:

```
CircleFigure circle = (CircleFigure)shape;
```

То же самое можно написать следующим образом:

```
Figure shape = new CircleFigure();  
CircleFigure circle = shape as CircleFigure;
```

Ключевое слово `as` указывает на то, что объект `shape` нужно воспринимать как экземпляр класса `CircleFigure`, каким он и является на самом деле.

У этих двух подходов есть существенное различие. Посмотрим на следующий пример:

```
Object obj = new int[10];
Console.WriteLine(obj as Shed);
Console.WriteLine((Shed)obj);
```

В первой строке создается массив из чисел и результат присваивается переменной. Это возможно, мы уже такое проходили.

Во второй строке в консоль выводится результат приведения к сараю. Так как переменная не является сараем, приведение невозможно, но результатом приведения будет `null`. Значит, в консоли мы ничего не увидим, — просто `null`.

А вот при использовании круглых скобок в случае невозможности приведения произойдет исключительная ситуация, и приложение упадет. С такими ошибками можно работать, и мы еще с этим познакомимся, но вот эту разницу нужно знать уже сейчас.

Получается, что с помощью `as` мы можем сделать проверку, такую же как и `is`:

```
Object obj = new int[10];
if ((obj as Shed) != null)
    // выполнить действие
```

По своей логике это практически `is`.

3.15. Неявный тип данных `var`

Начиная с .NET Framework 3.5, появился новый оператор: `var` (от слова *variable* — переменная). С его помощью можно создавать переменные без явного указания типа данных. Некоторые программисты, услышав такое заявление, думают, что Microsoft добавила уязвимость, потому что все в мире должно быть типизировано, иначе сложнее проверять корректность данных. Но .NET достаточно умная, чтобы определить тип данных по контексту инициализации.

Вы объявляете таким образом переменную, присваиваете ей значение, и платформа уже сама по контексту догадывается, какой нужно использовать тип:

```
var myVariable = 10;
```

Несмотря на то что мы явно не указали тип переменной `myVariable`, она все же автоматически станет числом, потому что компилятор способен догадаться об этом из контекста.

В .NET все же есть одно большое отличие от других интерпретируемых языков: тип данных в переменной, которая объявлена как `var`, не будет конвертирован автоматически в зависимости от контекста:

```
var numberVariable = 10;
var stringVariable = "Здесь будет строка";
```

Компилятор посмотрит на контекст и решит, что первая переменная явно число, и во время первого присвоения значения в переменную `numberVariable` будет зафиксировано, что здесь хранится число, а не что-либо другое.

А вот следующий код работать не будет:

```
var numberVariable = 10;
numberVariable = "Не работает";
```

В первой строке здесь объявляется переменная, которой присваивается число. Тут даже глупому компьютеру понятно, что 10 — это целое число, и тип данных будет, скорее всего, `Int`. Во второй строке происходит попытка записать в эту же переменную уже строку, но этот трюк не пройдет. Он сработает в JavaScript, PHP и многих (может быть, даже во всех, — я все языки не знаю) интерпретируемых языках, но не в .NET, потому что эта платформа требует строгой типизации, которая необходима для обеспечения безопасности.

Несмотря на то что оператор `var` очень соблазнительный и его так и хочется использовать везде, я очень не рекомендую этого делать. Это дело вкуса, но, на мой взгляд, такой код выглядит очень плохо, и его тяжело читать. Например, посмотрите на следующую строку кода:

```
var Person = GetFirstPerson();
```

и скажите мне, какого типа будет переменная `Person`. Я могу только подозревать, что есть какой-то класс `Person`, и объект этого класса возвращает метод `GetFirstPerson()`. Но это лишь благодаря тому, что имя переменной и имя метода на это указывают. А очень часто можно увидеть такой код:

```
var i = Calculate();
```

И здесь уже, глядя на код, ничего сказать нельзя. Понять, что делает этот код, можно только, если в Visual Studio воспользоваться контекстными подсказками и посмотреть на реальный код метода `Calculate`, но это лишь указывает на то, что подобный код не очень хороший. Опять же, это мое личное мнение, и достаточно большое количество программистов нормально относятся к подобному. Я же считаю, что код должен читаться без необходимости смотреть на контекст.

Написать реальный тип переменной не так уж и сложно, поэтому я всегда это делаю. И использую `var` только тогда, когда это действительно нужно.

3.16. Абстрактные классы

Иногда может возникнуть необходимость создать класс, экземпляры которого нельзя создавать. Например, вы хотите объявить класс фигуры, который будет хранить такие значения, как левая и правая позиции фигуры на форме, и ее имя. Вы так же можете объявить метод `Draw()`, который будет рисовать фигуру:

```
abstract class Figure
{
    public int left { get; set; }
    public int top { get; set; }
```

```
    abstract public void Draw();  
}
```

Ключевое слово `abstract` говорит о том, что нельзя создавать непосредственно объекты этого класса, и следующая строка кода завершится ошибкой:

```
Figure r1 = new Figure();
```

А зачем нужны абстрактные классы? Для того чтобы в них объявить какие-то свойства и методы, которые могут понадобиться в будущем другим классам. Например, в нашем случае можно создать два наследника: прямоугольник и круг, и возможная реализация этих классов показана в листинге 3.6.

Листинг 3.6. Наследование из абстрактного класса

```
class RectangleFigure : Figure  
{  
    public int Width { get; set; }  
    public int Height { get; set; }  
  
    public override void Draw()  
    {  
        Console.WriteLine("Это класс прямоугольника");  
    }  
}  
  
class CircleFigure : Figure  
{  
    public int Radius { get; set; }  
  
    public override void Draw()  
    {  
        Console.WriteLine("Это класс круга");  
    }  
}
```

Оба класса происходят от класса `Figure` (наследуют его), и мы можем создавать их объекты. Я скажу больше: мы можем создавать их как переменные класса `Figure`:

```
Figure rect;  
rect = new RectangleFigure();  
rect.Draw();  
  
rect = new CircleFigure();  
rect.Draw();
```

В этом примере объявляется переменная типа `rect`. Я специально объявил переменную в отдельной строке. Несмотря на то что класс `Figure` абстрактный, мы можем объявлять переменные такого типа, но не можем инициализировать их как `Figure`. Зато мы можем инициализировать эту переменную классом потока, что и происходит во второй строке.

Несмотря на то что переменная объявлена как `Figure`, в ней реально хранится `RectangleFigure`. Мы можем вызвать метод `Draw()` и убедиться, что в консоли появится сообщение, которое вызывает метод `Draw()` класса `RectangleFigure`. При этом мы не должны писать что-либо в скобках перед именем переменной `rect`, чтобы сказать, что перед нами класс `RectangleFigure`. Почему? Потому что у `Figure` есть метод `Draw()`, а благодаря полиморфизму и тому, что при переопределении мы использовали слово `override`, будет вызван метод именно того класса, которым проинициализирована переменная.

Получается, что абстрактные методы действуют точно так же, как виртуальные? Да, поэтому в потомках при переопределении мы можем использовать ключевое слово `override`.

После этого той же переменной присваивается экземпляр класса `CircleFigure`. Он тоже является наследником фигуры, поэтому операция вполне легальна. Если теперь вызвать метод `Draw()`, то на этот раз вызовется одноименный метод круга. Магия полиморфизма в действии!

Вот если бы класс `Figure` не содержал метода `Draw()`, нам пришлось бы явно говорить компилятору, что перед нами класс `RectangleFigure`, и можно использовать приведение типов:

```
(rect as RectangleFigure).Draw();
```

Когда мы пишем базовый класс и знаем, что всем наследникам понадобится какой-то метод (рисования, расчета площади или еще чего-нибудь), мы можем в базовом классе задать метод, но не реализовывать его. А вот наследники должны уже реализовать этот метод каждый по-своему. В предыдущем примере благодаря тому, что в базовом классе объявлен метод `Draw()`, мы можем вызывать этот метод одинаково как для круга, так и для прямоугольника, а в зависимости от того, какого класса объект находится в переменной, такой метод и будет вызван.

Если хорошо подумать, то возникают два вопроса: зачем в предке писать реализацию метода, когда он все равно должен быть переопределен в потомке, и как заставить потом переопределять метод. Когда есть возможность реализовать какой-то код по умолчанию, то его можно реализовать в предке, а наследники будут наследовать его и, если необходимо, переопределять. А вот когда наследник просто обязан переопределить метод, то метод можно сделать абстрактным.

В случае с нашим примером: что рисовать в методе `Draw()` класса `Figure`, когда мы не знаем, что это за фигура и какие у нее размеры? Метод `Draw()` тут бесполезен, и что-то в нем нереально, поэтому здесь желательно объявить метод как абстрактный:

```
abstract public void Draw();
```

Перед объявлением метода появилось слово `abstract`, а реализации тела метода нет вообще. Даже фигурные скобки отсутствуют. Теперь любой класс-потомок должен реализовать этот метод или должен быть тоже абстрактным, иначе компилятор выдаст ошибку.

Попробуйте создать проект с кодом наших фигур и объявить метод `Draw()` абстрактным. Теперь уберите реализацию метода `Draw()` из класса круга, и вы получите

ошибку компиляции: или реализуйте метод, или сделайте круг абстрактным. Можно выбрать второе, но тогда мы не сможем сделать экземпляры класса круга! Но мы можем создать наследника от круга `SuperCircle` и реализовать метод `Draw()` там, и тогда `SuperCircle` может быть не абстрактным, а его экземпляры можно будет создавать.

Если у класса есть хотя бы один абстрактный метод или хотя бы один из абстрактных методов, унаследованных от предка и не реализованных, то такой класс должен быть абстрактным.

3.17. Инициализация свойств

Во время создания объекта очень часто требуется задать множество свойств. Например, объект класса `Person` будет нуждаться в задании основных свойств — таких как имя, фамилия, дата рождения и т. д:

```
Person person = new Person();
person.FirstName = "Михаил";
person.LastName = "Фленов";
person.City = "Торонто";
person.Country = "Канада";
```

Можно создать конструктор, который будет принимать все нужные нам параметры и сразу же инициализировать, и я подобные подходы уже видел не раз, особенно в коде, написанном под первые версии .NET. Иногда это бывает правильным решением, но я не люблю, когда приходится передавать более трех параметров. Уж лучше потом задавать свойства, каждое в отдельности, как в только что приведенном примере.

Но в .NET появился способ сделать инициализацию немного проще, указав все свойства в фигурных скобках сразу после создания объекта:

```
Person person = new Person() {
    FirstName = "Михаил",
    LastName = "Фленов",
    City = "Торонто",
    Country = "Канада"
}
```

В чем смысл, брат? Наверное, такой вопрос зародился у вас, потому что с точки зрения компактности мы практически ничего не выиграли — все так же нужно писать большое количество имен свойств и задавать им значения. Смысл в том, что все это один оператор.

Допустим, нам нужно создать объект и передать его в метод. И вот тут компактный способ инициализации подходит просто великолепно:

```
MethodCall(
    new Person() {
        FirstName = "Михаил",
```

```
        LastName = "Фленов",  
        City = "Торонто",  
        Country = "Канада"  
    }  
};
```

Здесь я проинициализировал и передал новый объект методу `MethodCall` в один подход.

3.18. Частицы класса

При проектировании классов их нужно создавать такими, чтобы они выполняли только одну задачу. Если следовать этому простому правилу, ваши классы будут небольшими и легко читаться в одном файле.

Но бывают редкие случаи, когда мы должны добавить один метод к уже определенному классу или когда код содержит два ярко выраженных кода.

Ко второму случаю я отношу классы, которые будут содержать работу с визуальным интерфейсом. В случае с визуальным интерфейсом очень удобно для одного класса создать два файла с исходным кодом. В одном из файлов будет код создания визуального окна и компонентов, а во втором файле — код, реагирующий на различные события: нажатия кнопок, выбор меню и т. д.

Такое можно реализовать через частичные (*partial*) классы. Давайте создадим класс `MyForm` с одним методом в файле `MyForm.cs`:

```
public partial class MyForm {  
    void Method1() {  
    }  
}
```

А еще часть класса может находиться совершенно в другом файле — например, `MyFormEvents.cs`:

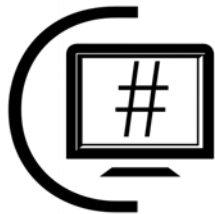
```
public partial class MyForm {  
    void Method2() {  
    }  
}
```

Теперь оба метода: `Method1` и `Method2` — являются частью одного класса `MyForm`. Если теперь создать экземпляр класса `MyForm`, то мы можем задействовать оба метода:

```
MyForm form = new MyForm();  
form.Method1();  
form.Method2();
```

Несмотря на то что методы были объявлены в разных файлах, они — часть одного целого класса, и мы их можем использовать так же, как будто никакого разделения на файлы не было.

ГЛАВА 4



Консольные приложения

Большинство читателей, наверное, уже очень сильно хотят приступить к созданию интересных программ. Но я все же рекомендую вам потратить еще немного времени и изучить консольный мир чуть ближе, потому что он тоже интересен и может заворочить вас своей простотой и конкретностью.

Зачем нужна консоль в нашем мире визуальности? Некоторые считают, что консоль должна умереть как пережиток прошлого. Она была необходима, когда компьютеры были слабенькими и не могли потянуть сложные и насыщенные графикой задачи. Зачем же снова возвращаться к текстовым командам, разве что лишь в целях обучения?

Причин, почему в компьютерах до сих пор широко используется консоль, есть множество, и самая главная из них — это сеть. Консольные программы выполняются в текстовом режиме, и с ними очень удобно работать, когда администратор подключается к серверу удаленно через программы текстового терминала. В таких случаях у администратора есть только командная строка и нет графического интерфейса.

Чтобы получить графический интерфейс при подключении к удаленному серверу, необходимо намного больше трафика и намного больше ресурсов, поэтому командная строка до сих пор жива и даже развивается. Так, в Windows Server 2008 появилась командная оболочка PowerShell, которая позволяет из командной строки выполнять практически любые манипуляции с системой. Эта же оболочка доступна и для Windows 7/8/10/11, только скачивать и устанавливать ее придется отдельно.

Последние годы я в основном работаю с веб-приложениями, но при этом мне регулярно приходится взаимодействовать с консолью. Например, если кто-то на сайте покупает товар, то после покупки, скорее всего, будет задействовано много консольных программ, которые по расписанию выполняют запросы к базе данных, чтобы найти все новые заказы, выгрузить и перенаправить заказы на склад, чтобы там начали собирать продукт. Очень часто консольные приложения занимаются и отправкой почты.

Еще одна причина, по которой эта глава необходима начинающим именно сейчас, — она сгладит переход от основ, которые мы изучали до этого, к более сложным примерам.

Консольные приложения для .NET (которые выполняются на любой платформе) и консольные приложения для .NET Framework под Windows идентичны. А в этом случае стоит выбирать .NET, потому что эти приложения можно будет выполнить на Linux или macOS.

Как уже отмечалось в *предисловии*, в четвертое издание книги я добавил материал по работе с .NET Core, поэтому все исходники для этой главы были переписаны с использованием шаблона консольного приложения .NET Core. Позднее Microsoft поменяла название фреймворка на .NET 5, но это все еще потомок версии Core. Именно такое приложение мы и создавали в *разд. 1.3.1*. Исходные коды для .NET Framework тоже никуда не делись, и вы их можете найти в электронном архиве, сопровождающем книгу. Для этого в электронный архив материалов для *главы 4* включены две папки:

- ❑ Sources/Chapter4 — исходные коды для .NET Framework;
- ❑ Sources/Chapter4.Core — исходные коды для версии .NET 5, которые работают идентично ее предшественнику .NET Core.

Если вы сравните эти исходные коды, то увидите, что они практически идентичны, поэтому в следующих далее примечаниях про электронный архив упоминается только папка Chapter4.

4.1. Украшение консоли

Консоль в Windows — это класс `Console` определенного типа окна, и у него есть несколько свойств, которые позволяют управлять этим окном и параметрами текста в нем. Давайте посмотрим на свойства, которые вы можете изменять для настройки консоли, оформления и просто для работы с ней.

Наверное, самое популярное оформительское свойство — это цвет текста: `ForegroundColor`. Тут нужно заметить, что почти везде цвет в .NET описывается классом `Color`, но в консоли цвет в виде исключения описан как `ConsoleColor`. И если `Color` — это класс, то `ConsoleColor` — перечисление (объявлено в системе как `public enum ConsoleColor`), которое содержит не такое уж и большое, но вполне достаточное количество цветов.

Чтобы узнать, какие цвета доступны, можно открыть MSDN, а можно в редакторе кода набрать `ConsoleColor` и нажать клавишу с точкой (без пробелов). В ответ должен появиться выпадающий список с доступными значениями (рис. 4.1). Если выпадающий список не появился, попробуйте поставить курсор сразу за точкой и нажать комбинацию клавиш `<Ctrl>+<Пробел>`.

Следующая строка показывает, как можно изменить цвет текста в консоли на зеленый:

```
Console.ForegroundColor = ConsoleColor.Green;
```

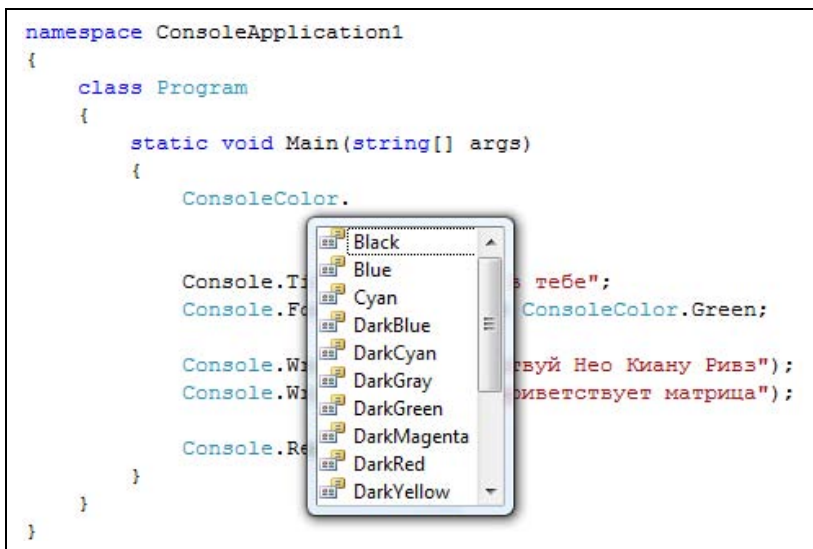


Рис. 4.1. Выпадающий список с возможными значениями ConsoleColor в редакторе кода

Тут нужно отметить, что цвет текста уже выведенных в консоль сообщений не изменится. В новых цветах засияет только весь последующий текст, который вы будете вводить в консоль. Например:

```

// меняю цвет текста на зеленый
Console.ForegroundColor = ConsoleColor.Green;
// следующие две строки текста будут зелеными
Console.WriteLine("Здравствуй, Нео Киану Ривз.");
Console.WriteLine("Тебя приветствует матрица!");
// меняю цвет текста на красный
Console.ForegroundColor = ConsoleColor.Red;
// следующие две строки текста будут красными
Console.WriteLine("Здравствуй, Тринити.");
Console.WriteLine("Признавайся, где Морфеус!");

```

На рис. 4.2 показан результат работы программы. Черно-белая печать не сможет передать всей красоты примера, но все же вы должны заметить, что первые две строки отличаются по цвету от последних двух.

Следующее свойство: `BackgroundColor` — определяет цвет фона, на котором выводится текст, и имеет тип `ConsoleColor`. Обратите внимание, что изменяется цвет фона только под текстом, а не всего окна консоли. Если в предыдущем примере надо между зеленым и красным текстом изменить цвет фона на желтый, то следует написать так:

```
Console.BackgroundColor = ConsoleColor.Yellow;
```

А что если вы захотите изменить цвет всего окна? Это возможно, просто после изменения цвета фона нужно очистить окно консоли, для чего используется метод

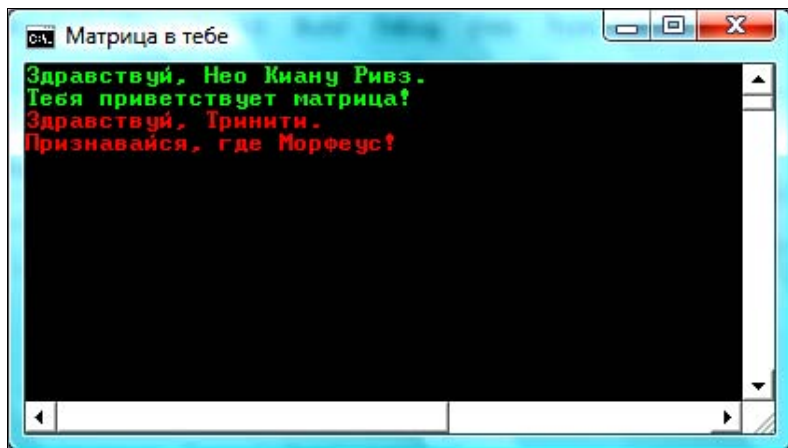


Рис. 4.2. Эффект изменения цвета текста

`Clear()`. Например, следующий пример изменяет цвет фона текста на белый и очищает консоль:

```
Console.BackgroundColor = ConsoleColor.White;  
Console.Clear();
```

Свойство `CapsLock` имеет тип `bool` и возвращает `true`, если нажата клавиша <Caps Lock>. Это свойство только для чтения, поэтому вы можете не пытаться изменить его, — ничего, кроме ошибки компиляции, не увидите. Следующие две строки кода проверяют, нажата ли клавиша <Caps Lock>, и если да, то выводится соответствующее сообщение:

```
if (Console.CapsLock)  
    Console.WriteLine("Отключите <Caps Lock>.");
```

Свойство `NumberLock` позволяет определить, нажата ли клавиша <Num Lock> в текущий момент. Если свойство вернет `true`, то клавиша нажата.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter4\Configure` сопровождающего книгу электронного архива (см. приложение).

4.2. Работа с буфером консоли

Если вы думаете, что консоль — это просто текстовое окно, в которое можно лишь последовательно выводить информацию, то вы ошибаетесь. Консоль — это целый буфер с памятью, куда данные могут выводиться даже хаотично. Вы можете переводить курсор по буферу, а также создавать что-то типа ASCII-графики. Я в такой графике не силен, поэтому не смогу вам показать супермегапримеры, но постараюсь сделать что-то интересное.

Обратите внимание, что когда окно консоли запущено, то справа появляется полоса прокрутки. Это потому, что буфер строк по умолчанию очень большой и рассчитан

на 300 строк. Буфер колонок (символов в ширину) предусматривает всего 80 символов, поэтому горизонтальной прокрутки нет. Но если уменьшить окно, то появится и горизонтальная полоса прокрутки. Чтобы узнать размеры буфера, можно воспользоваться свойствами: `BufferHeight` (высота буфера) и `BufferWidth` (ширина буфера) класса `Console`. Оба значения возвращают количество символов соответствующего буфера.

С помощью свойств `CursorLeft` и `CursorTop` вы можете узнать или изменить позицию курсора относительно левой и верхней границ буфера соответственно. Давайте посмотрим на пример, который выводит приблизительно в центре окна (если оно имеет размеры по умолчанию) названия допустимых к использованию в консоли цветов:

```
ConsoleColor[] colors = { ConsoleColor.Blue, ConsoleColor.Red,
    ConsoleColor.White, ConsoleColor.Yellow };
foreach (ConsoleColor color in colors)
{
    Console.CursorLeft =
        (Console.BufferWidth - color.ToString().Length) / 2;
    Console.CursorTop = 10;
    Console.ForegroundColor = color;
    Console.WriteLine(color);
    Thread.Sleep(1000);
    Console.Clear();
}
```

Этот очень интересный пример красив не только тем, что он отображает, но и тем, что и как он это делает. Перед циклом создается массив из нескольких значений цветов, доступных для консоли. Затем запускается цикл `foreach`, который просматривает весь этот массив. В принципе, то же самое и даже круче можно было бы сделать в одну строку, но просто не хочется сейчас обсуждать вопросы, выходящие за рамки главы. Впрочем, если вам это интересно, то следующая строка кода запустит цикл, который переберет абсолютно все значения цветов из перечисления `ConsoleColor`:

```
foreach (ConsoleColor color in
    Enum.GetValues(typeof(ConsoleColor)))
```

Но вернемся к предыдущему примеру. Внутри цикла в первой строке я устанавливаю левую позицию так, чтобы сообщение выводилось посередине окна. Для этого из ширины буфера вычитается ширина строки с именем цвета и делится пополам. Тут самое интересное — процесс определения размера имени цвета. Цвет у нас имеет тип `ConsoleColor`, но если вызвать метод `ToString()`, то мы получаем имя цвета в виде строки. А вот у строки есть свойство `Length`, в котором находится размер строки. В качестве отступа сверху выбираем 10 символов. У меня это примерно середина окна, если его размеры не трогать.

Теперь можно изменить цвет текста на текущий, чтобы мы визуально заметили это, и вывести его название. Обратите внимание, что консольному методу `WriteLine()` передается переменная `color`, которая имеет тип `ConsoleColor`, и мы не вызываем

явно метод `ToString()`. Дело в том, что если нужно привести тип к строке, то метод `ToString()` вызывается автоматически.

После вывода текста в консоль вызывается строка `Thread.Sleep(1000)`. Пока что не будем углубляться в ее разбор, а только запомним, что она устанавливает задержку на количество миллисекунд, указанных в круглых скобках. В одной секунде 1000 миллисекунд, а значит, наш код сделает задержку в секунду. Запомнили? Закрепили? Поехали дальше.

В .NET Core нет класса `Thread`, и там эту строку нужно заменить на:

```
System.Threading.Tasks.Task.Delay(1000).Wait();
```

Последняя строка вызывает метод `Clear()`, чтобы очистить консоль. Для чего это нужно? Дело в том, что если название следующего выводимого в центре экрана цвета окажется короче предыдущего, то оно не сможет его затереть, и все символы и остатки старого названия останутся. Попробуйте убрать эту строку и запустить пример, чтобы убедиться в его необходимости.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter4\ConsoleBuffer* сопровождающего книгу электронного архива (см. приложение).

4.3. Окно консоли

Свойство `Title` класса `Console` — текстовая строка, которая отображает заголовок окна. Вы можете изменять его по своему желанию. Следующая строка кода программно изменяет заголовок окна консоли:

```
Console.Title = "Матрица в тебе";
```

Свойства `WindowHeight` и `WindowWidth` позволяют задать высоту и ширину окна соответственно. Значения задаются в символах и зависят от разрешения экрана. Для моего монитора с разрешением 1280×1024 максимальной высотой оказалось число 81. Если указать большее число, то команда завершается ошибкой, поэтому нужно или отлавливать исключительные ситуации, чего мы пока делать не умеем, или не нагнать. Я предпочитаю вообще не трогать размеры окна, чтобы не поймать лишнюю исключительную ситуацию. Хотя на самом деле я очень редко пишу консольные программы.

Свойства `WindowLeft` и `WindowTop` позволяют задать левую и верхнюю позиции окна относительно экранного буфера. Тут нужно быть внимательным, потому что это не позиция окна на рабочем столе, а позиция в экранном буфере. Изменяя эту позицию, вы как бы программно производите прокрутку.

4.4. Запись в консоль

Пока что самый популярный метод в этой книге, который мы применяли уже множество раз, но скоро перестанем (потому что перейдем к использованию визуального интерфейса), — `WriteLine()`. Метод умеет выводить на экран текстовую стро-

ку, которая передается в качестве параметра. Мы чаще всего передавали одну строку — по крайней мере, я старался делать так, чтобы не загружать вам голову лишней информацией. Сейчас настало время немного напрячься, тем более что у нас уже достаточно знаний, чтобы этот процесс прошел максимально плавно и безболезненно.

Итак, что же такого интересного в методе `WriteLine()`? Дело в том, что существует аж 19 перегруженных вариантов этого метода — на все случаи жизни. Большинство из них — это просто вариации на тему типа данных, получаемых в параметре. Метод может принимать числа, булевы значения и т. д., переводить их в строку и выводить в окно консоли. В общем-то, ничего сложного, но есть один вариант метода, который заслуживает отдельного упоминания, и мы сейчас предоставим ему такую честь.

Вот несколько вариантов метода, которые будут нас интересовать:

```
WriteLine(String, Object);  
WriteLine(String, Object, ...);  
WriteLine(String, Object[]);
```

Во всех приведенных здесь вариантах метода первый параметр — это строка форматирования. Что это такое? Это просто текст, внутри которого могут быть указаны определенные места, в которые нужно вставить некие аргументы. Эти места указываются в виде `{xxx}`, где `xxx` — это, опять же, не фильм с Вином Дизелем, а номер аргумента. Сами аргументы передаются во втором, третьем и т. д. параметрах. Например, посмотрим на следующую строку:

```
Console.WriteLine("Хлеб стоит = {0} рублей", 25);
```

В первом параметре указана строка, в которой есть ссылка на нулевой аргумент: `{0}`. Нулевой аргумент — это второй параметр, а значит, мы увидим в результате на экране: `Хлеб стоит = 25 рублей`.

В строке форматирования вы можете использовать специальные символы форматирования, например:

- `\n` — переход на новую строку;
- `\r` — переход в начало строки.

Так, следующая команда выводит с помощью одного метода сразу две строки. Специальный символ `\n` будет заменен на переход на новую строку:

```
Console.WriteLine("Строка 1\nСтрока 2");
```

Еще усложним задачу и выведем одной командой два числа — каждое в своей строке:

```
Console.WriteLine("Это число {0}\nЭто еще число {1}", 10, 12);
```

В результате в консоли будет:

```
Это число 10  
Это еще число 12
```

Когда аргументов много, то их удобнее передавать через массив. Следующий пример выполняет почти такую же задачу, что и предыдущая строка, только все аргументы группируются в массив и передаются во втором параметре:

```
Console.WriteLine(
    "Это число {0}\nДругое число {1}\nИ снова первое число {0}",
    new Object[] { 3, 4 }
);
```

Есть еще более интересный метод форматирования: {XXX:F}, где XXX — это все тот же индекс аргумента, а F — символ форматирования, который может быть одним из следующих:

- c — аргумент является денежной единицей. Следующий код:

```
Console.WriteLine("Хлеб стоит {0:c}", 25);
```

для русских региональных настроек ОС Windows выведет на экран: Хлеб стоит 25,00р. Как видите, система сама добавила дробную часть для копеек и сокращение денежной единицы.

По умолчанию система будет выводить деньги в формате, который указан в свойствах системы (**Панель управления | Язык и региональные стандарты**). Но вы можете управлять количеством нулей после запятой. Если вы хотите работать с банковской точностью (в банках и курсах валют очень часто используют четыре знака после запятой), то после буквы c укажите необходимое количество таких символов. Например, следующий пример отобразит цену хлеба с 4 символами после запятой:

```
Console.WriteLine("Хлеб стоит {0:c4} ", 25);
```

- d — аргумент является простым десятичным числом. После символа форматирования можно указать минимальное количество символов, которые должны быть напечатаны. Недостающие символы будут добавлены нулями. Код:

```
Console.WriteLine("Хлеб стоит {0:d5}", 25);
```

даст в результате 00025;

- f — выводит аргумент с определенным количеством символов после запятой. Код:

```
Console.WriteLine("Хлеб стоит = {0:f3}", 25.4);
```

выведет на экран 25,400;

- n — группирует числа по разрядам. Например, если формат {0:n} применить к числу 25000, то на выходе мы получим 25 000;

- e — аргумент должен быть выведен в экспоненциальном виде;

- x — выводит значение аргумента в шестнадцатеричном формате.

Для вывода информации в консоль есть еще один метод: Write(). Он отличается от WriteLine() только тем, что после вывода в консоль не переводит текущую позицию ввода на новую строку. То есть два следующих вызова абсолютно идентичны:

```
Console.Write("Строка" + "\n");  
Console.WriteLine("Строка");
```

В первом случае я добавил переход на новую строку явно с помощью символа `\n`, а во втором случае он будет добавлен автоматически.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter4\ConsoleMethods* сопровождающего книгу электронного архива (см. *приложение*).

4.5. Чтение данных из консоли

Для чтения из консоли существуют два метода: `Read()` и `ReadLine()`. Первый метод читает данные посимвольно, т. е. при обращении к методу он возвращает очередной символ из потока ввода, по умолчанию — из окна консоли.

Все это выглядит следующим образом. Когда пользователь вводит строку в консоль и нажимает клавишу `<Enter>`, то метод возвращает код первого из введенных символов. Второй вызов метода вернет код второго символа. И так далее. Обратите внимание, что метод возвращает числовой код, а не символ в виде типа данных `char`. Чтобы получить символ, нужно конвертировать число в тип данных `char`.

Давайте посмотрим использование этого метода на примере:

```
char ch;  
do  
{  
    int x = Console.Read();    // чтение очередного символа  
    ch = Convert.ToChar(x);    // конвертирование в тип char  
    Console.WriteLine(ch);    // вывод символа в отдельной строке  
} while (ch != 'q');
```

Здесь запускается цикл `do...while`, который будет выполняться, пока пользователь не введет символ `q`. Внутри цикла первой строкой вызывается метод `Read()`, который читает символы из входного потока. В этот момент программа застынет в ожидании ввода со стороны пользователя. Когда пользователь введет строку и нажмет клавишу `<Enter>`, строка попадет в определенный буфер, и из него уже по одному символу будет читаться в нашем цикле с помощью метода `Read()`.

Следующей строкой кода мы конвертируем код прочитанного символа непосредственно в символ. Это осуществляет класс `Convert`, имеющий статичный метод `ToChar()`. Этот метод умеет легким движением процессора превращать индекс символа в символ, который и возвращается в качестве результата. Далее мы выводим прочитанное на экран в отдельной строке, и если прочитанный символ не является буквой `q`, то цикл продолжается.

Тут нужно заметить, что выполнение программы прервется не только после того, как пользователь введет отдельно стоящую букву `q`, а после любого слова, в котором есть эта буква. То есть после ввода слова `faq` программа выведет два первых его символа `i`, увидев `q`, завершит выполнение цикла.

Запустите программу и введите какое-нибудь слово. По нажатию клавиши <Enter> программа выведет все буквы введенной фразы — каждую в отдельной строке, и в конце добавит еще две строки. Откуда они взялись? Дело в том, что в ОС Windows нажатие клавиши <Enter> состоит аж из двух символов — с кодами 13 и 10 (именно в такой последовательности они будут добавлены в конец передаваемого программе буфера): конец строки и возврат каретки. Они невидимы, но при разборе строки посимвольно эти символы будут видны программе. Забегая вперед, скажу, что метод `WriteLine()` не видит этих символов.

Чтобы избавиться от пустых строчек из-за невидимых символов, можно добавить:

```
if (x != 10 && x != 13)
    Console.WriteLine(ch);
```

Кстати, символ с кодом 13 на самом деле соответствует специальному символу `\n`, а символ с кодом 10 соответствует `\r`. Мы уже использовали такие символы в этой главе ранее, когда нужно было добавить в середину строки разрыв.

Метод `ReadLine()` тоже читает данные из входного буфера, но он возвращает буфер в виде строки целиком. Следующий пример показывает, как можно читать данные из консоли, пока пользователь не введет букву `q`:

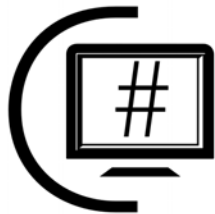
```
string str;
do
{
    str = Console.ReadLine();
    Console.WriteLine(str);
} while (str != "q");
```

На этот раз для выхода вам нужно ввести в строке только `q` и ничего больше, потому что для выхода из цикла мы ищем не просто символ `q` во введенной строке, а сравниваем полученную строку целиком с буквой `q`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter4\ConsoleRead` сопровождающего книгу электронного архива (см. приложение).

ГЛАВА 5



Продвинутое программирование

Когда мы с вами изучали основы C#, мне пришлось опустить очень много интересных вопросов, потому что мы тогда еще не знали, что такое *классы*. Сейчас мы сможем погрузиться в изучение C# и программирование под платформу .NET на более интересных примерах.

В этой главе я иногда могу использовать при объявлении переменных префиксы — первая буква имени будет указывать на тип данных переменной. В реальных проектах я так не делаю, но в книге это имеет смысл ради большего удобства и ясности.

5.1. Приведение и преобразование типов

Мы уже немного познакомились с приведением и преобразованием типов, а сейчас настало время свести все знания в одно целое. Приведение типов бывает явным и неявным. При *неявном* мы просто обращаемся к переменной, а она приводится к другому типу. Например:

```
int i = 10;  
object obj = i;
```

В первой строке здесь создается числовая переменная, а во второй строке объектной переменной `obj` присваивается значение числовой переменной. Процедура превращения одной переменной в другую происходит автоматически и незаметно для глаза программиста, когда нет потери данных. На самом деле тут даже нет никакого преобразования, это просто магия объектно-ориентированного программирования.

В C# все типы данных — это *объекты*, но если и числа всегда обрабатывать как объекты, то это будет очень дорого для платформы. Простые типы данных, такие как числа или логические переменные, обрабатываются специальным образом. Они могут восприниматься компилятором как простые типы данных, но как только мы обращаемся к ним как к объекту, они воспринимаются платформой как объекты. Этот процесс называется упаковкой (boxing) и распаковкой (unboxing). Более подробно об этом рассказано в *разд. 5.2*.

Пока же отметим, что *упаковка* — это помещение простого типа в обертку объекта, после чего вы можете работать с этим числом как с объектом. Я рассматриваю этот процесс сейчас, потому что отчасти это можно отнести к преобразованию.

Неявного преобразования в C# практически нет — везде нужны явные действия, иначе компилятор начнет «ругаться». Там, где вам кажется, что происходит неявное преобразование, чаще всего просто используется какой-то перегруженный метод.

Самый распространенный способ приведения типов — написать перед переменной в скобках имя нового типа. Например:

```
int c = (int)obj;
```

Это именно *приведение* типов, потому что тут не происходит преобразования объекта `obj` в числовую переменную. Приведение типов — это когда в переменной хранится число, но нам передали его в виде объекта `object`. Приведением мы просто показываем, что в объектной переменной на самом деле хранится число. При этом не происходит никакого преобразования значения переменной из одного в другой — для приведенного примера это как раз и называется *распаковкой*.

Если в `obj` будет находиться не число, то произойдет ошибка. То есть мы должны быть уверены, что переменная имеет нужный нам тип, а с помощью типа данных в скобках мы просто говорим компилятору, чтобы он воспринимал `obj` как число.

Если нужно выполнить именно *преобразование*, то можно использовать класс `Convert`. Этот класс содержит множество статических методов для различных типов данных. Например, если вы хотите превратить какую-то переменную в тип даты и времени `DateTime`, то нужно использовать метод `ToDateTime()`:

```
Тип_Данных переменная;  
DateTime dt = Convert.ToDateTime(переменная);
```

Что принимает этот метод? У него 18 разных перегруженных вариантов для 18 различных типов, поэтому он может принимать логические значения, числа, строки и т. д., и он будет пытаться любой из этих типов привести к типу `DateTime`.

Если нам нужно привести что-либо к строке `string`, то на этот случай класс `Convert` имеет метод `ToString()`, у которого тоже есть 18 перегруженных вариантов. Как вы думаете, какой метод будет использоваться для конвертирования в число `Int32`? Конечно же, `ToInt32()`.

Класс `Convert` и его методы не просто указывают, что нужно использовать какую-то переменную как тот или иной тип данных (приведение), — они выполняют именно преобразование. А это уже более мощный механизм. Есть еще один способ преобразования, но его мы рассмотрим в *разд. 5.2*.

На самом деле приводить к строке любой тип данных проще всего. Дело в том, что у класса `Object` есть метод `ToString()`, который возвращает объект в виде строки. А раз он есть у `Object`, значит, есть и у всех остальных классов, потому что все классы обязательно содержат среди предков класс `Object`. Большинство классов

переопределяют этот метод, чтобы он возвращал нормальное значение для того или иного типа. Если вы создаете свой класс и не переопределяете метод `ToString()`, то он вернет тип данных в виде строки.

Теперь посмотрим на следующий код:

```
int i = 10;
string s = "" + i;
```

Что произойдет во второй строке кода, где мы складываем текст с числовой переменной? Тут совершится неявное для нас, но явное для компилятора преобразование типов. Переменная `i` будет преобразована к строке с помощью вызова метода `ToString()`, а далее осуществится уже банальная конкатенация строк. Поскольку нужно привести тип к строке, а любой тип можно без проблем привести к строке с помощью `ToString()`, то среда разработки выполнения и компилятор достаточно интеллектуальны, чтобы преобразовывать тип автоматически. Подобный метод мы уже много раз использовали и будем использовать в дальнейшем.

Обратное же превращение строки к числу подобным способом невозможно. Для этого следует использовать явное преобразование с помощью `Convert`.

5.2. Все в .NET — это объекты

Мы уже говорили, что в .NET все типы данных являются объектами. Попробуйте объявить переменную типа `int`. Теперь напишите эту переменную и нажмите точку. Должен появиться выпадающий список, в котором приведены свойства и методы текущего объекта (рис. 5.1). Если он не появился, то поставьте курсор после точки и нажмите магическую комбинацию клавиш `<Ctrl>+<Пробел>` или `<Ctrl>+<J>`.

Так что, тип данных `int` является не простым типом данных, а на самом деле представляет собой объект? Почти так. Простые типы данных тоже происходят от объекта, но помимо этого еще и от `ValueType`.



Рис. 5.1. Методы простого типа данных `int`

Когда нужно произвести с простыми данными математические операции, то система воспринимает их как простые типы данных. Если бы это было не так, мы не смогли бы складывать числа простым знаком сложения. Я даже представить себе не могу, как можно универсально сложить два объекта. Для каждого класса сложение может выполняться по-разному. Но если обратиться к переменной как к объекту, то система автоматически упакует значение в объект, и мы увидим методы, как у класса.

Чтобы удобнее было работать с типами данных как с классами, в .NET существуют и специализированные типы данных — их имена начинаются с большой буквы: `Int16`, `Int32`, `Int64`, `Double` и т. д. У этих классов есть множество статических методов, которые вы можете использовать для различных операций. Например, следующий код объявляет строковую переменную, а во второй строке происходит преобразование с помощью статического метода `Parse()` класса `Int32`:

```
string sNumber = "10";  
int iNumber = Int32.Parse(sNumber);
```

Методу `Parse()` нужно передать строку, а он на выходе вернет число. При работе с методом надо быть аккуратным, потому что, если строка содержит некорректные данные, которые не могут быть приведены к числу, произойдет ошибка выполнения. Более безопасным является вызов метода `TryParse()`:

```
string sNumber = "10";  
int iNumber;  
if (Int32.TryParse(sNumber, out iNumber))  
    Console.WriteLine(iNumber);
```

Метод `TryParse()` имеет два параметра:

- строку, которую нужно перевести в число;
- числовую переменную с ключевым словом `out`, через которую мы получим результат.

А что же тогда возвращает метод? Он возвращает булево значение. Если оно равно `true`, то строку удалось превратить в число, иначе строка содержит некорректные данные, которые невозможно перевести в число.

В .NET фреймворке очень часто можно увидеть статические методы у классов для простых типов данных — таких как `Int32`, с помощью которых можно производить преобразования.

Возможно, я сейчас скажу самую странную вещь, но простые типы данных на самом деле не классы, а *структуры*. Да, `Int32` наследуется от `ValueType`, который является классом, а тот, в свою очередь, наследуется от `Object`, а в результате мы получаем структуру.

Что такое структуры? О них будет рассказано в *разд. 5.4*, а пока просто продолжайте воспринимать простые данные как простые типы значений, а то, что это структура, — не меняет правил игры. Структуры в C# — это тоже типы, значения которых обрабатываются в памяти одним способом.

5.3. Работа с перечислениями *Enum*

Когда мы рассматривали перечисления в *разд. 2.4.3*, то я упустил несколько интересных моментов, и это упущение мы сейчас восполним. Я сделал это намеренно, потому что тогда мы еще не знали про классы и не могли «скакать через их голову».

5.3.1. Перечисления — это значения

Прежде всего мы должны понять, что перечисления на самом деле это числовой тип данных. По умолчанию для их хранения будет использоваться тип данных `int`, но можно указать и любой другой числовой тип данных. Тип указывается после имени перечисления и до фигурных скобок, как это делается при наследовании классов.

В следующем примере для перечисления используется беззнаковое целое число `unsigned int`:

```
public enum MyColors : uint { Red, Green, Blue };
```

Перечисления могут объявляться как вне класса, так и внутри объявления класса. В первом случае перечисление станет доступно для всех классов соответствующего пространства имен, а при наличии модификатора `public` и всем сторонним классам. Такое перечисление будет выступать в роли самостоятельной единицы, и любой класс сможет объявить переменную этого типа перечисления.

Если же перечисление объявлено как часть класса, то доступ к перечислению по умолчанию будет только у этого класса. А если поставить модификатор доступа `public`, то и другие классы тоже смогут ссылаться на это перечисление, но не как на самостоятельную единицу, а как на член класса. Например, если перечисление `MyColors` объявлено внутри класса `Form`, то в классе `Test` переменная типа перечисления `MyColors` будет объявляться по полному имени `Form.MyColors`, которое включает имя класса, внутри которого находится объявление:

```
namespace EnumIndex
{
    public partial class Form
    {
        public enum MyColors { Red, Green, Blue };
        ...
    }
    public class Test
    {
        Form1.MyColors myTestColor;
    }
}
```

С перечислениями тоже можно работать как с объектами. Для этого в .NET существует класс `Enum` (именно так — с заглавной буквы), у которого имеется ряд весьма полезных статических методов.

Откройте исходный код любого проекта или создайте новый проект для теста и добавьте объявление перечисления `MyColors`, содержащего три цвета:

```
enum MyColors
{
    Red = 100,
    Green = 200,
    Blue = 300
};
```

Я задал именам перечисления еще и индексы просто для красоты примера, вы же можете задать любые другие значения или оставить их по умолчанию. Если возникает вопрос, где написать объявление: внутри класса (сделать перечисление членом класса) или вне его (чтобы сделать его самостоятельной единицей), я бы порекомендовал сделать его членом класса, потому что к этому перечислению мы будем обращаться только из класса `Program`, а точнее — из метода `Main` класса `Program`.

Теперь в конструкторе после вызова метода `Main()` добавьте следующий код:

```
Console.WriteLine("Enum Names");

foreach (string str in Enum.GetNames(typeof(MyColors)))
    Console.WriteLine(str);

Console.WriteLine("Enum Values");
foreach (int i in Enum.GetValues(typeof(MyColors)))
    Console.WriteLine(i);
```

В результате мы должны будем увидеть в консоли:

```
Enum Names
Red
Green
Blue
Enum Values
100
200
300
```

Здесь запускаются два цикла: первый — перебирает все имена, которые есть в перечислении, а второй — все индексы перечисления. Чтобы получить массив всех имен перечисления, можно воспользоваться статическим методом `GetNames()` класса `Enum`. В качестве параметра ему надо передать тип данных для перечисления. Как получить этот тип данных? Интересный вопрос. Для этого служит оператор `typeof`.

Оператор `typeof` чем-то похож на метод, потому что он тоже принимает в качестве параметра какое-то значение (в нашем случае — перечисление, но это может быть и класс) и возвращает значение (здесь — тип переменной). Тем не менее есть и различия — оператор не принадлежит какому-то классу, он просто существует как оператор присваивания, деления, умножения и т. д.

Итак, `typeof(MyColors)` вернет нам тип данных для перечисления. По этому типу статичный метод `GetNames()` класса `Enum` вернет массив строк, в котором находятся имена, входящие в перечисление. Нам только остается с помощью цикла `foreach` пересмотреть все имена. В нашем случае внутри цикла мы банально выводим в консоль имя.

Теперь второй цикл понять намного проще. Он также перебирает все элементы массива, но только числового, который будет возвращен статичным методом `GetValues()`. Этот метод также получает тип перечисления, а возвращает значения элементов перечисления.

Следующая задача: допустим, что вы получаете от пользователя имя и должны превратить его в перечисление. Мы пока работаем с консолью, поэтому предположим, что имя вводится в консоль, и мы читаем его:

```
Console.WriteLine("Введите имя цвета");  
string currentColor = Console.ReadLine();
```

Теперь в строковой переменной `currentColor` находится имя цвета, которое мы ввели, и нам надо превратить его в перечисление типа `MyColors`:

```
MyColors myColor1 =  
    (MyColors)Enum.Parse(typeof(MyColors), currentColor);  
Console.WriteLine("Вы выбрали" + myColor1 + " - " + (int)myColor1);
```

В *разд. 5.2* для превращения строки в число мы использовали `Int32.Parse`, а для перечислений вполне логично было бы иметь что-то типа `Enum.Parse`. Так и есть. Для преобразования строки в значение представления мы можем использовать метод `Enum.Parse`.

Этот метод получает два параметра: тип перечисления, который мы можем получить уже знакомым нам способом (`typeof(MyColors)`), и строку с именем. Тип необходим, потому что `Enum.Parse` не может знать, перечисление какого именно типа мы передадим, — это может быть что угодно. Если мы передадим строку `Green`, то это может быть перечисление `MyColors` или что-то другое, — мало ли кто создал `Enum` с таким именем.

Получив объект перечисления, я вывожу его значение, но чуть-чуть другим методом:

```
Console.WriteLine("Вы выбрали" + myColor1 + " - " + (int)myColor1);
```

На этот раз я вывожу в строку имя и значение перечисления несколько иначе — через интерполяцию строк. Использование `myColor1` в строке — это то же самое, что написать `myColor1.ToString()`, и оно приведет к отображению на экране имени.

После этого я использую приведение к числу `(int)myColor1`, и в результате будет отображено значение.

Если запустить этот код и ввести корректное значение — например, `Green`, то мы должны будем увидеть в результате в консоли:

```
Вы выбрали Green - 200
```


Но нужно вводить `Green` именно в том же регистре, что и в объявлении. Если ввести все слово маленькими буквами, то произойдет ошибка, потому что приведение типов станет невозможным. Можно использовать немного другую версию, которая принимает три параметра, где третьим параметром идет булево значение, указывающее, нужно ли игнорировать регистр:

```
Enum.Parse(typeof(MyColors), currentColor, true);
```

В этом случае будет игнорироваться регистр, но все еще может произойти ошибка, если кто-то введет совсем неверную строку.

Можно использовать `TryParse`:

```
object myColorObject;  
if (Enum.TryParse(typeof(MyColors), currentColor, true,  
    out myColorObject))  
{  
    myColor1 = (MyColors)myColorObject;  
    Console.WriteLine("Вы выбрали" + myColor1 + " - " +  
        (int)myColor1);  
}
```

А что если пользователь вводит число и нам нужно превратить его в перечисление? Если пользователь указал `100`, то это должно превратиться в `MyColors.Red`:

```
Console.WriteLine("Введите значение цвета");  
currentColor = Console.ReadLine();  
int colorIntValue = Int32.Parse(currentColor);  
MyColors myColor2 = (MyColors)colorIntValue;  
Console.WriteLine("Вы выбрали " + myColor2 + " - " + (int)myColor2);
```

На этот раз я прошу пользователя ввести число, потом использую `Int32.Parse` для получения числового значения и, наконец, превращаю это число в представление — и это делается банальным приведением типов: `(MyColors)colorIntValue`.

Что если пользователь введет нереальное значение — например, `123`? Такого цвета нет, но ошибки во время приведения не произойдет. А какое тогда имя будет отображаться? Имени не будет, будет число:

```
Вы выбрали 123 - 123
```

5.3.2. Перечисления-флаги

А что если мы захотим хранить в переменной несколько дней недели? Например, рабочие дни недели. Можно завести строку и хранить в ней имена дней недели:

```
string workdays = "Monday, Tuesday, Wednesday";
```

Но со строкой работать неудобно — как определить, что в строке есть понедельник? Чтобы узнать, есть ли среди рабочих дней понедельник, нужно будет искать по строке, а поиск по строке — не самая быстрая операция. Не самая медленная, но в то же время и не очень быстрая.

Намного удобнее и лучше использовать *флаги* на базе перечислений. Для этого заводим перечисление, а в качестве значений указываем разные биты:

```
public enum Weekdays {  
    Monday    = 0b_0000_0001,  
    Tuesday   = 0b_0000_0010,  
    Wednesday = 0b_0000_0100,  
    Thursday  = 0b_0000_1000,  
    Friday    = 0b_0001_0000,  
    Saturday  = 0b_0010_0000,  
    Sunday    = 0b_0100_0000  
};
```

Так как значения перечисления заданы как биты в разных позициях, то для работы с ними лучше использовать бинарные операции. Например, в следующем примере заводится переменная для хранения выходных и в нее записываются два дня: суббота и воскресенье:

```
Weekdays weekend = Weekdays.Saturday | Weekdays.Sunday;  
Console.WriteLine("Weekend: " + weekend);
```

В результате в консоли вы должны увидеть:

```
Weekend: 96
```

Разбираемся, почему 96. Суббота — это значение перечисления, у которого 6-й бит равен единице, а остальные — нули. 2 в 6-й степени — это 32. Воскресенье — это двойка в седьмом бите, т. е. 64. Даже если математически сложить 32 и 64 простым плюсом, то мы получим 96. А если сложить их логически:

```
0b_0010_0000  
ИЛИ  
0b_0100_0000
```

то результатом будет число, в котором единице будут равны те биты, в которых 1 имеется в первом *ИЛИ* во втором числе:

```
0b_0110_0000
```

Если это число перевести в десятичную систему, то снова будет 96.

Подождите, раз логическая операция *ИЛИ* будет давать такой же результат, что и сложение, так, может, использовать привычное нам математическое сложение?

```
weekend = (int)Weekdays.Saturday + Weekdays.Sunday;  
Console.WriteLine("Weekend: " + weekend);
```

Такой код возможен, и он будет работать отлично и даже даст нам все то же самое число 96. Но его использование опасно, потому что если дважды сложить субботу и один раз воскресенье:

```
weekend = (int)Weekdays.Saturday + (int)Weekdays.Sunday +  
    Weekdays.Saturday;  
Console.WriteLine("Weekend: " + weekend);
```

то результатом будет 128, но это же бред...

А если сделать логическое сложение:

```
weekend = Weekdays.Saturday | Weekdays.Saturday | Weekdays.Sunday;  
Console.WriteLine("Weekend: " + weekend);
```

результатом будет 96, потому что:

```
0b_0010_0000  
ИЛИ  
0b_0010_0000  
ИЛИ  
0b_0100_0000
```

В результате мы получим число, в котором единице будут равны те биты, в которых 1 равен хотя бы один байт, а это все то же число:

```
0b_0110_0000
```

Операция **ИЛИ**, примененная к одному и тому же числу, его не меняет:

```
0b_0010_0000  
ИЛИ  
0b_0010_0000
```

Получим:

```
0b_0001_0000
```

Выводить в консоль числа может быть не очень хорошо. Чтобы в консоли появились имена, достаточно перед объявлением перечисления добавить [Flags]:

```
[Flags]  
public enum Weekdays {
```

Теперь в консоли мы должны увидеть вместо числа 96 имена перечислений:

```
Weekend: Saturday, Sunday
```

Как теперь проверить, есть ли среди выходных воскресенье? Для этого используется логическая операция **И**:

```
0b_0011_0000  
И  
0b_0010_0000
```

Первое число — это 96 или наша переменная `weekend`, а второе число — это воскресенье. При операции **И** результатом будет число, в котором 1 находится в одной и той же позиции, — т. е. мы должны получить: `0b_0010_0000`.

А если проверить, есть ли среди выходных понедельник:

```
0b_0011_0000  
И  
0b_0000_0001
```

Здесь нет ни одного совпадающего байта, равного единице, а значит, результатом будет ноль. Получается, что мы можем проверять флаги с помощью логического сложения и потом проверять, равен результат нулю или нет:

```
(weekend & Weekdays.Monday) != 0  
(weekend & Weekdays.Sunday) != 0
```

В первом случае результатом будет `false`, а во втором — `true`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter5\EnumTest` сопровождающего книгу электронного архива (см. приложение).

5.4. Структуры

Структуры — самый интересный тип, потому что он может работать и как простой тип (без инициализации с помощью оператора `new`), но вроде бы выглядит как ссылочный тип. .NET просто прячет от нас всю инициализацию, и структуры выглядят как простые типы данных — такие же, как числа.

Возможно, я вас сейчас немного испугаю, но в C# тип данных `Int32` на самом деле — структура. Мне кажется, это сделано для простоты, потому что с точки зрения управления памятью числа работают так же, как и структуры.

Но для начала разберемся, что такое *структура*. Это набор типов данных, сгруппированных под одним именем. Допустим, нам нужно описать программно человека. Он характеризуется следующими параметрами: имя, фамилия, возраст. Для хранения этой информации можно создать три переменные, и это будет нормально, пока у нас один человек, а если у нас их сто?

Когда требуется описать много людей, можно создать три массива, в каждом из которых будет по одному атрибуту человека: имя, фамилия и возраст. А что если будет десяток атрибутов? Положение спасают структуры. Мы просто объединяем атрибуты человека в структуру и создаем массив структур.

Объявление структуры очень похоже на описание класса — к членам структуры допускается указывать модификаторы доступа, может также существовать и конструктор, в котором можно задавать значения по умолчанию:

```
struct Person  
{  
    public Person(string firstName, string lastName)  
    {  
        FirstName = firstName;  
        LastName = lastName;  
        age = 18;  
    }  
  
    public string FirstName;  
    public string LastName;  
    public int age;  
}
```

До 7-й версии C# конструкторы обязаны были инициализировать все поля структуры. Если нет конструктора, то создается конструктор по умолчанию, который это делает автоматически, но если вы сами создаете конструктор, то раньше он должен был явно инициализировать все поля, однако, начиная с 7-й версии C#, так делать уже не обязательно.

Давайте посмотрим, как теперь использовать структуру в коде. Ранее уже отмечалось, что структура может выступать как простой тип данных, а значит, мы ее можем просто объявить как простую переменную и использовать без выделения памяти с помощью оператора `new`:

```
Person p;  
p.FirstName = "Сергей";  
Console.WriteLine(p.FirstName);
```

Это вполне корректный пример, в котором объявляется переменная `p` типа `Person`. Во второй строке кода изменяется имя в структуре, а в третьей строке это имя выводится в окно консоли. Так как мы не выделяли память, конструктор структуры не вызывался, и все его поля (переменные) равны нулю. Запомните это, это очень важно!

А что если попробовать вывести в консоль значение переменной `p.LastName`? В чем тут соль? Мы ее не изменяли и не устанавливали, а раз конструктор не вызывался, то переменная равна нулю:

```
Console.WriteLine(p.LastName);
```

Что произойдет? Произойдет ошибка уже на этапе компиляции. Когда вы попытаетесь собрать исполняемый файл, среда разработки сообщит об ошибке: **Use of possibly unassigned field 'LastName'**, что примерно означает: «Возможно использование неназначенного поля». Получается, что если переменная типа структуры объявлена как простая переменная, то ее поля можно использовать только после явного присвоения значения конкретному полю. Именно в этот момент будет происходить инициализация, но не всей структуры, а только значения конкретного поля.

А вот если инициализировать структуру через оператор `new`, то и все переменные будут проинициализированы:

```
Person p1 = new Person();  
Console.WriteLine("Фамилия 1: " + p1.LastName);
```

В этом примере сразу после создания структуры я пытаюсь вывести ее в консоль, и операция пройдет успешно. Просто переменная пустая, и на экране не появится фамилия, но и ошибки не будет.

Следующий пример инициализирует структуру с помощью конструктора, который мы прописали так:

```
Person p2 = new Person("Михаил", "Фленов");  
Console.WriteLine("Фамилия 2: " + p2.LastName);
```

Конструктор принимает начальные значения для имени и фамилии, и переданные значения будут записаны в соответствующие поля структуры.

У структур могут быть даже методы, которые будут выполнять какие-то действия:

```
struct Person
{
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
        Age = 18;
    }

    public string FirstName;
    public string LastName;
    public int Age;

    public void Print() {
        Console.WriteLine("First Name " + FirstName);
        Console.WriteLine("Last Name " + LastName);
        Console.WriteLine("Age " + Age);
    }
}
```

Получается, у структур есть свойства, методы, конструкторы, и тогда чем они отличаются от классов? Самая основная разница кроется в том, где выделяется память для хранения этого типа данных. Я как-то говорил, что для простых данных память выделяется в стеке, потому что для них известен точный размер. Для динамических данных (классов, массивов и т. д.) память выделяется в куче, или в динамической памяти.

Так вот, память для структур выделяется в стеке. Это значит, что если мы будем использовать структуру как объект, то потребуется упаковка и распаковка, что потребует лишних действий со стороны процессора. Это минус. Зато с точки зрения управления памятью структуры лучше, потому что стек освобождается более эффективно и не требует затрат на сборщик мусора.

Если вам придется использовать данные с методами, которые требуют объект, то, скорее всего, выгоднее будет использовать класс.

Здесь структура объявлена в качестве поля класса:

```
class Test {
    public Person Employee { get; set; }
}
```

То есть создается класс со свойством `Employee`, которое имеет тип структуры `Person`. В этом случае, когда будет выделяться в куче память, тут же будет выделена и память для этого свойства, — объект будет хранить данные и структуры. Чтобы освободить память, достаточно уничтожить только объект класса `Test`.

Если же `Person` объявить классом, то при создании объекта в нем не будет данных свойства. Для хранения объекта `Employee` будет выделена отдельная память в куче,

и в объекте класса `Test` станет храниться только ссылка на `Employee`. Таким образом в памяти кучи будут находиться два объекта, и сборщик мусора должен будет их отслеживать и уничтожать по мере освобождения. Это дольше и сложнее.

Так, может, всегда создавать структуры, если они проще с точки зрения управления памятью?

Структуры проще с точки зрения управления памятью, но сложнее с точки зрения передачи данных.

Давайте отметим еще одно очень важное отличие структур от классов. Посмотрите на следующий код:

```
Person p = new Person("Михаил", "Фленов");
Foo(p);
Console.WriteLine(p.FirstName);

void Foo(Person p)
{
    p.FirstName = "updated";
}
```

Здесь я создал переменную типа `Person`, вызвал метод `Foo` и вывожу после этого в консоль свойство `FirstName`. Внутри метода изменяется свойство `FirstName` на другое.

Внимание, вопрос: что будет, если `Person` и мы выполним этот код? В консоли будет выведено Михаил. Потому что при вызове методов происходит копирование параметра. Структура — это значение, и хранится в памяти как значение, а значит, перед вызовом будет создана полная копия переменной `p`, и эта копия будет передана внутрь метода. Любые изменения внутри метода будут происходить уже над копией, и никак не повлияют на оригинал.

Если у структуры 100 различных свойств и вы будете ее передавать в другие методы, то .NET может потратить слишком много времени на создание копий всех 100 свойств. Если даже каждое свойство будет числом, придется скопировать 100 чисел.

А что если мы работаем с классом?

```
PersonObj pobj = new PersonObj() {
    FirstName = "Михаил"
};
Foo(pobj);
Console.WriteLine(pobj.FirstName);

void Foo(PersonObj p)
{
    p.FirstName = "updated";
}
```

```
class PersonObj
{
    public string FirstName;
}
```

В этом случае `pobj` становится ссылкой на данные в куче, и перед вызовом `Foo` будет создана копия ссылки, которая станет указывать на один и тот же объект. А значит, при использовании объектов в консоли мы увидим `Updated`.

Если мы хотим разделить объект между кодом и методом, то нужно создавать объекты, и тогда мы будем работать с одними и теми же данными. Сами ссылки по размеру очень маленькие, и их копирование не отнимает много ресурсов.

И напоследок еще один маленький нюанс:

```
void Foo(PersonObj p)
{
    p = new Person();
}
```

Внутри этого метода я пытаюсь поменять не свойство `p`, а саму ссылку. Но т. к. это копия оригинальной ссылки, эта замена не повлияет на оригинал. Вы можете менять свойства объекта `p`, потому что это будут одни и те же данные с оригиналом. Да, две разные ссылки указывают на один и тот же объект, но изменение самой ссылки изменяет копию, а не оригинал.

Последнее небольшое отличие классов от структур заключается в том, как сравниваются две структуры и два класса. Заведем два разных экземпляра структуры и сравним их с помощью метода `Equals`:

```
Person ps1 = new Person("Михаил", "Фленов");
Person ps2 = new Person("Михаил", "Фленов");
Console.WriteLine("ps1 == ps2 " + ps1.Equals(ps2));
```

Результатом будет истина. Несмотря на то что здесь два разных экземпляра, они равны, потому что структуры сравниваются по значениям всех свойств. Если все свойства идентичны, то структуры идентичны.

А если создать два объекта и сравнить их:

```
PersonObj po1 = new PersonObj() { FirstName = "Михаил" };
PersonObj po2 = new PersonObj() { FirstName = "Михаил" };
Console.WriteLine("po1 == po2 " + po1.Equals(po2));
```

результатом будет ложь. Две переменные объектов будут идентичны, только если они указывают на один и тот же объект. Если объекты разные, `Equals` будет возвращать ложь, даже если все свойства идентичны.

Таким образом, если вам нужно хранить одно или два свойства как единое целое, то их может быть удобно хранить как структуру. Копирование при вызове методов не отнимет много времени, и можно будет сэкономить на сборщике мусора. Если же полей много и хранится много данных, то лучше создавать класс.

А если нужно описать точку в двумерном пространстве, то, скорее всего, лучше использовать структуру:

```
struct Point
{
    public int x { get; set; }
    public int y { get; set; }
}
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter5\StructProject* сопровождающего книгу электронного архива (см. приложение).

5.5. Дата и время

Для работы датой и временем в C# используются структуры `DateTime` и `TimeSpan`. Для хранения даты нужно не так много памяти, поэтому не составит труда копировать ее при вызове методов и выбор структуры очевиден.

Структура `DateTime` позволяет работать с определенной датой и временем.

Так как `DateTime` — это структура, то ее нужно инициализировать с помощью оператора `new`. У `DateTime` есть 12 различных перегруженных конструкторов на все случаи жизни. Вы можете создать объект даты, у которого будет задана только дата без времени, можете указать и то и другое. Объект может быть построен по различным составляющим и информации о дате.

Для следующего примера я написал небольшой статичный метод, чтобы мы могли читать с консоли числа:

```
static int GetNumber(string message)
{
    while (true) {
        Console.WriteLine("Введите " + message);
        string input = Console.ReadLine();
        int number;
        if (Int32.TryParse(input, out number))
        {
            return number;
        }
        Console.WriteLine("неверное значение");
    }
}
```

Метод в бесконечном цикле отображает в консоли приглашение ввести число, и если пользователь вводит строку, которую нельзя преобразовать в число, то отображается ошибка и запрос повторяется. Если пользователь ввел корректное число, то `GetNumber` возвращает его.

Теперь одной строкой мы можем прочитать число с консоли:

```
int year = GetNumber("год");
```

Используя этот метод, я читаю из консоли пять значений: год, месяц, день, час, минуты, создаю объект `DateTime` из полученных данных и отображаю день недели, часы, время:

```
static void Main(string[] args)
{
    int year = GetNumber("год");
    int month = GetNumber("месяц");
    int day = GetNumber("день");
    int hour = GetNumber("час");
    int minute = GetNumber("минуты");
    DateTime dt = new DateTime(year, month, day, hour, minute, 0);
    Console.WriteLine("День недели: " + dt.DayOfWeek.ToString());
    Console.WriteLine("День недели #: " + ((int)dt.DayOfWeek).ToString());
    Console.WriteLine("Время: " + dt.TimeOfDay.ToString());
    Console.WriteLine("День: " + dt.DayOfYear.ToString());
    Console.WriteLine(".ToString: " + dt.ToString());
    Console.WriteLine("date.ToString: " + dt.Date.ToString());
    Console.WriteLine(".ToShortDateString: " + dt.ToShortDateString());
    Console.WriteLine(".ToLongDateString: " + dt.ToLongDateString());
    Console.WriteLine(".ToShortTimeString: " + dt.ToShortTimeString());
    Console.WriteLine(".ToLongTimeString: " + dt.ToLongTimeString());
    Console.ReadLine();
}
```

Недостаток этого кода в том, что он читает с консоли число, но если на запрос на ввод месяца пользователь введет число 21, то это приведет к ошибке, поскольку мы не проверяем, больше ли 12 введенное число.

Сначала только ради наглядности я сохраняю в целочисленных переменных значения, которые ввел пользователь для составляющих даты. После этого инициализирую объект `DateTime`. Для этого я выбрал конструктор, который получает в качестве параметров компоненты даты и времени, начиная с года и до секунды. Но т. к. я не запрашивал у пользователя секунды, то эту составляющую я просто обнуляю, передавая явно число 0.

Далее идет более интересный код. Имея переменную типа `DateTime`, мы можем многое узнать о дате и времени. Например, какой день недели выпал на ту злополучную или знаменательную дату, которая сохранена в переменной. Для этого можно воспользоваться свойством `DayOfWeek`. Именно это мы и узнаем в первой строке кода после инициализации объекта.

Свойство `DayOfWeek` — это перечисление, подобное тому, что мы создавали сами, а раз в Америке неделя начинается с воскресенья, то, значит, под индексом 0 будет воскресенье, а понедельник будет иметь значение 1.

По умолчанию приведение перечисления к строке вернет нам название в виде текста. Чтобы получить строку, достаточно привести название к числу следующим образом:

```
((int)dt.DayOfWeek).ToString()
```

Обратите внимание на скобки. Они так расставлены далеко не случайно. Если опустить крайние скобки и написать просто:

```
(int)dt.DayOfWeek.ToString()
```

то к типу `int` будет приводиться вся конструкция `dt.DayOfWeek.ToString()`, т. е. имя дня недели. Это невозможно сделать, и компилятор выдаст ошибку. Нам нужно привести к числу день недели `dt.DayOfWeek`, поэтому крайними скобками я как раз и указываю на это. А вот результат приведения к числу превращается в строку с помощью метода `ToString()`.

А что если нам нужно решить классическую задачу: узнать, сколько сейчас времени? Для этого у `DateTime` есть статическое свойство `Now`, с помощью которого эта задача и решается. Следующая строка кода инициализирует переменную `dt` текущим значением даты и времени:

```
DateTime dt = DateTime.Now;
```

Обратите внимание, что мы не вызываем никаких конструкторов, а просто присваиваем переменной класса `DateTime` значение свойства `Now`. Это вполне корректно, потому что свойство само создает новый экземпляр объекта `DateTime` и возвращает его нам, поэтому в дополнительной инициализации нет необходимости.

При работе с датами очень часто приходится выполнять математические операции над их составляющими. `DateTime` имеет множество методов, с помощью которых удобно изменять значение даты или времени:

- ☐ `AddYears(int N)` — добавить к дате `N` лет;
- ☐ `AddMonths(int N)` — добавить `N` месяцев;
- ☐ `AddDays(int N)` — добавить `N` дней;
- ☐ `AddHours(int N)` — добавить `N` часов;
- ☐ `AddMinutes(int N)` — добавить `N` минут;
- ☐ `AddSeconds(int N)` — добавить `N` секунд;
- ☐ `AddMilliseconds(int N)` — добавить `N` миллисекунд.

Все эти методы возвращают результат, а не изменяют содержимое объекта. Чтобы изменить значение переменной, нужно написать так:

```
dt = dt.AddDays(-60);
```

В этом случае из даты в переменной `dt` будет вычтено 60 дней. Да, чтобы произвести вычитание, нужно указать в качестве параметра отрицательное значение. И еще — вы не ограничены размерностью составляющей, которую изменяете. Это значит, что если вы изменяете количество дней, то можно смело писать 60 дней и не

нужно вычислять, сколько это целых месяцев, и вычитать месяцы и дни по отдельности. Методы класса `DateTime` очень умные и все подсчитают сами.

Есть еще один метод, с помощью которого можно изменять значение объекта класса `DateTime`, — это `Add()`. Метод получает в качестве параметра значение типа `TimeSpan`, а это значит, что пришлось время разобраться с этим классом.

Класс `TimeSpan` — это интервал времени. Ему все равно, начиная с какой и по какую дату длится интервал, — он просто хранит значение. Например, `TimeSpan` может быть равен 15 минутам. Это просто 15 минут.

Интервалы хорошо использовать для вычислений и работы с датами. Например, следующие строки кода увеличивают дату на 15 минут:

```
TimeSpan ts = new TimeSpan(0, 15, 0)
dt = dt.Add(ts);
```

В первой строке создается интервал времени. В качестве параметров конструктор `TimeSpan` в нашем случае принимает часы, минуты и секунды. Существуют и другие перегруженные конструкторы — например, принимающие составляющие, начиная с количества дней и до секунд. Максимальная размерность интервала — дни, но количество дней может равняться и 100. Во второй строке кода мы просто вызываем метод `Add()` объекта `DateTime`, чтобы добавить интервал.

Этот пример удобнее было бы записать в одну строку:

```
dt = dt.Add(new TimeSpan(0, 15, 0));
```

Здесь методу `Add()` передается новая структура, проинициализированная из `TimeSpan`. Это то же самое, что было раньше, просто теперь мы не сохраняем значение интервала в переменной, а сразу передаем его методу `Add()`.

Любая составляющая может быть отрицательной. Например, следующий код создаст интервал, равный −1 час и +10 минут:

```
TimeSpan ts = new TimeSpan(-1, 10, 0);
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter5\TimeSpanProject` сопровождающего книгу электронного архива (см. приложение).

5.6. Класс строк

Все это время мы работали со строками достаточно поверхностно — просто присваивали значения и производили операцию конкатенации. Но это только самая малость из того, что может понадобиться программисту в реальной работе. В этом разделе мы познакомимся со строками чуть ближе и узнаем о нескольких очень интересных методах класса `String`:

- `Contains()` — позволяет узнать, содержит ли строка подстроку, переданную в качестве параметра. Если да, то метод вернет `true`;

- ❑ `Format()` — это статичный метод, который позволяет форматировать строки, как мы это делали в консольных приложениях, описанных в *главе 4*. Внутри строки можно в фигурных скобках указывать места, куда должны вставляться переменные, переданные во втором и т. д. параметрах. Так как метод статичный, то мы вызываем его через класс, а не через объект:

```
String str;  
Str = String.Format("Приветствие миру '{0}'", "Hello, world");
```

- ❑ `IndexOf()` — возвращает индекс символа, начиная с которого в строке найдена подстрока, переданная в качестве параметра. Если ничего не найдено, то результатом будет `-1`. Например, следующая строка кода ищет в строковой переменной `str` текст `"world"`:

```
int index = str.IndexOf("world");
```

- ❑ `Insert()` — позволяет вставить подстроку, переданную во втором параметре, в строку, начиная с символа, указанного в первом параметре. Следующий пример вставляет слово `" мир"` в переменную `str`, начиная с 5-й позиции:

```
string newstr = str.Insert(5, " мир");
```

При этом переменная `str` не изменяется, а новая строка просто возвращается в виде результата;

- ❑ `PadLeft(int N)` и `PadRight(int N)` — эти методы очень похожи, потому что их задача: вписать строку в новую строку определенного размера. Размер передается в качестве параметра. Методы создают строку указанного размера и вписывают в нее строку так, чтобы она была выровнена по одному из краев.

Так, при использовании метода `PadLeft()` строка будет размещена справа, а слева будет добавлено столько пробелов, чтобы в результате получилась строка длиной `N`. Использование метода `PadRight()` добавляет к строке пробелы справа до длины `N`;

- ❑ `Remove()` — удаляет из строки символы, начиная с индекса, указанного в качестве первого параметра, и ровно столько символов, сколько указано во втором параметре. Если во втором параметре ничего не задано, то удаление происходит до конца строки. Как всегда, сама строка не изменяется, измененный вариант возвращается в качестве результата:

```
str = str.Remove(2, 3); // удалить 3 символа, начиная со второго
```

- ❑ `Replace()` — ищет в строке подстроку, указанную в качестве первого параметра, и заменяет ее на подстроку из второго параметра, возвращая результат замены. Следующий пример заменяет `"world"` на `"мир"`:

```
str = str.Replace("world", "мир");
```

- ❑ `ToUpper()` и `ToLower()` — методы возвращают строку, в которой все символы приведены к верхнему (`ToUpper()`) или к нижнему (`ToLower()`) регистру;

- ❑ `Substring()` — возвращает часть строки, начиная с символа, указанного в качестве первого параметра, и ровно столько символов, сколько указано во втором параметре;
- ❑ `ToCharArray()` — превращает строку в массив символов. Метод очень удобен, когда нужно проанализировать строку посимвольно, — например, следующий код получает массив символов, а потом перебирает его с помощью цикла `foreach`:

```
char[] chars = str.ToCharArray();  
foreach (char ch in chars)  
    Console.WriteLine(ch);
```

А что если внутри строки нужно поставить двойную кавычку?

```
"Hello, "world""
```

Такой код не скомпилируется, потому что непонятно, где начало/конец строки, а где кавычки внутри строки. Вместо этого нужно использовать управляющие символы, и для двойной кавычки это `\`:

```
"Hello, \"world\""
```

Вот теперь понятно, что если кавычка идет после слеша, то это часть строки.

Внутри строк вы можете использовать следующие управляющие коды:

- ❑ `\'` — вставить одинарную кавычку;
- ❑ `\a` — иницирует системный сигнал;
- ❑ `\n` — переход на новую строку;
- ❑ `\r` — возврат каретки;
- ❑ `\t` — символ табуляции.

Так как символ обратного слеша управляющий, то, чтобы добавить его в строку, нужно его удвоить. Это значит, что для задания пути к файлу в строке надо использовать двойные слеша:

```
string path = "c:\\windows\\system32\\filename.txt";
```

Если вы не хотите удваивать слеша и не собираетесь использовать управляющие коды, то перед строкой можно поставить символ `@`:

```
string path = @"c:\windows\system32\filename.txt";
```

Внутри такой строки управляющие коды работать не станут, а будут выводиться буквально, без интерпретации. Под управляющими символами имеется в виду не только слеш, но и символы новой строки.

До сих пор мы заключали строки в двойные кавычки, и чтобы поместить в одну переменную две строки, приходилось разделять их с помощью управляющего символа `\n`, как в следующем примере:

```
string multyline1 = "Строка 1\nСтрока 2";  
Console.WriteLine("multyline 1 = " + multyline1);
```

Когда перед строкой есть символ @, то управляющий символ \n работать не будет, но мы можем просто использовать несколько строк:

```
string multiline1 = @"Строка 1
    Строка 2";
Console.WriteLine("multiline 1 = " + multiline1);
```

Очень часто бывает необходимо проверить, является ли строка пустой или нулевой. Это можно сделать несколькими способами, и первый из них — в лоб проверять на null и на пустое значение:

```
String s = Get userInput();
if (s == null || s == "") {
    пользователь ничего не ввел
}
```

Подобные проверки очень часто приходится делать в веб-программировании, когда мы ожидаем ввода от пользователя.

То же самое можно сделать в один прием с помощью метода IsNullOrEmpty:

```
String s = Get userInput();
if (String.IsNullOrEmpty(s)) {
    пользователь ничего не ввел
}
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter5\StringProject* сопровождающего книгу электронного архива (см. приложение).

5.7. Перегрузка операторов

Согласитесь, что выполнять математические операции над простыми типами данных очень удобно. А что если у вас есть класс, который представляет что-либо математическое, и вы хотите работать с ним так же, как и с простыми типами, используя операторы сложения, вычитания, умножения и т. д. В C# это вполне возможно, только вам самим нужно позаботиться о том, как будут производиться математические вычисления.

Операторы связаны не только с математикой, но и с логикой. Есть еще операторы сравнения, которые тоже хотелось бы использовать с некоторыми классами.

Но самое мощное и интересное решение — *операторы приведения типов*. До сих пор мы умели приводить только совместимые типы данных. Например, мы могли привести классы к их предкам, но не могли приводить один класс к другому, если у них нет ничего общего и они выведены из разных предков. А с помощью операторов приведения типов можно сделать так, чтобы человека можно было привести даже к классу точки, и это далеко не полный бред.

5.7.1. Математические операторы

Начнем мы с самого простого, на мой взгляд, с математических операторов. Допустим, у нас есть класс `MyPoint`, который описывает точку (листинг 5.1).

Листинг 5.1. Класс точки

```
class MyPoint
{
    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    int x;
    public int X
    {
        get { return x; }
        set { x = value; }
    }

    int y;
    public int Y
    {
        get { return y; }
        set { y = value; }
    }

    public override string ToString()
    {
        return x.ToString() + ":" + y.ToString();
    }
}
```

Как было бы прекрасно создать два объекта точки, а потом сложить их простым оператором сложения:

```
MyPoint p1 = new MyPoint(10, 20);
MyPoint p2 = new MyPoint(20, 10);
MyPoint p3 = p1 + p2;
```

С первыми двумя строками проблем нет, а вот с третьей возникнут серьезные проблемы, и компилятор выдаст ошибку: **Operator '+' cannot be applied to operands of type MyPoint and MyPoint**, что означает примерно следующее: «Оператор '+' не может быть использован с типами данных `MyPoint` и `MyPoint`». Как культурно он нас послал!

А давайте попробуем встать на место компилятора. Допустим, вы видите операцию сложения двух объектов (не имеет значения, каких). Как вы их будете складывать? Простое сложение всех открытых полей не является хорошим решением, потому что не все может складываться. Это мы сейчас, глядя на объявление `MyPoint`, видим, что нужно просто сложить свойства `x` и `y`, а если у этого класса будет еще и свойство с именем точки (свойство `Name`), — что делать с ним? Тоже складывать? А компилятор может дать гарантию, что именно это нам нужно? Нет, он не может быть настолько умным.

Если компилятор точно не знает, чего мы от него хотим, то он не станет этого делать. Так происходит и с операторами. Мы должны явно в своем классе реализовать нужные операторы, которыми хотим пользоваться. Вы можете перегрузить почти все операторы языка C# (+, -, *, /, %, &, |, ^, <<, >>), а также операторы сравнения (==, !=, <, >, <= и >=). Определение операторов похоже на методы, только вместо имени метода нужно указать ключевое слово `operator` и указать оператор, который вы хотите определить. И, кроме того, такие определения должны быть статичными, ведь они вызываются не для конкретного объекта, а просто вызываются.

Рассмотрим, например, как могут быть реализованы операторы сложения и вычитания для наших точек:

```
public static MyPoint operator + (MyPoint p1, MyPoint p2)
{
    return new MyPoint(p1.X + p2.X, p1.Y + p2.Y);
}

public static MyPoint operator - (MyPoint p1, MyPoint p2)
{
    return new MyPoint(p1.X - p2.X, p1.Y - p2.Y);
}
```

Действительно, эти объявления сильно похожи на методы. В скобках указываются два складываемых параметра. Первый из них стоит слева от знака оператора, а второй — справа. Возвращаемое значение в нашем случае имеет тип `MyPoint`, но это не обязательно. Вы можете написать оператор сложения так, что он будет создавать в результате объект класса `int`. Как говорят американцы, «the sky is the limit» (в смысле: нет предела совершенству), т. е. вы можете создавать совершенно разные операторы на свое усмотрение.

Код оператора прост — нужно создать новый экземпляр класса и вернуть его. При этом он должен быть суммой (в вашем представлении) двух переданных объектов.

Вот теперь сложение двух объектов класса `MyPoint` пройдет без проблем, и следующая строка будет корректна для компилятора:

```
MyPoint p3 = p1 + p2;
```

А что если мы захотим складывать дом и точку? Да без проблем, просто добавляем следующий оператор к нашему классу точки:

```
public static MyPoint operator +(House h1, MyPoint p2)
{
    return new MyPoint(h1.Width + p2.X, h1.Height + p2.Y);
}
```

Теперь мы можем сложить дом и точку:

```
House house = new House(20, 10);
MyPoint p3dsum = house + p3;
```

Тут есть один нюанс, о котором мы уже говорили, — дом в операторе объявлен в качестве первого параметра, а значит, он должен находиться слева от знака сложения. Если поставить дом справа, а объект точки слева, то произойдет ошибка.

Напоследок замечу, что сокращенные операторы (такие как `+=`, `-=`, `*=` и `/=`) переопределять не нужно. Достаточно только определить операторы сложения, вычитания, умножения и деления, которые получают в качестве параметров типы `MyPoint` и возвращают тип `MyPoint`.

Не стоит реализовывать абсолютно все операторы во всех классах. Реализуйте только те операторы, которые действительно нужны и имеют смысл. Например, для нашей точки можно реализовать, наверное, все операторы, но если перед нами находится класс `Person`, то тут нужно задуматься, стоит ли реализовывать операторы умножения и деления и имеют ли они какой-то смысл? Ну разве что сложение — если сложить мужской и женский пол, то с задержкой в 9 месяцев можно попытаться получить новый объект класса `Person`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter5\MyPointProject` сопровождающего книгу электронного архива (см. приложение).

5.7.2. Операторы сравнения

Теперь давайте поговорим об *операторах сравнения*. Мы уже знаем, что для сравнения на равенство компилятор использует метод `Equals()`, который наследуется от базового класса `Object`. Мы уже научились переопределять и использовать его, но как обстоят дела со сравнением на больше или меньше? Тут компилятор и среда исполнения также не могут догадаться, как сравнивать классы, потому что они не обладают телепатическими способностями. Мы сами должны реализовывать операторы сравнения.

Как сравнить две точки на плоскости? Существует несколько вариантов сравнения, и все зависит от того, что представляет собой точка. Это может быть график, у которого ось `x` имеет приоритетное значение, и тогда нужно сравнивать координаты `x` двух точек, и если они равны, то сравнивать координаты `y`. Мы же поступим примитивно — сложим координаты `x` и `y` каждой точки и сравним результат:

```
public static bool operator < (MyPoint p1, MyPoint p2)
{
    return (p1.X + p1.Y < p2.X + p2.Y);
}
```

```
public static bool operator > (MyPoint p1, MyPoint p2)
{
    return (p1.X + p1.Y > p2.X + p2.Y);
}
```

Операторы всегда возвращают булево значение (`bool`), ведь результат сравнения — это логическая переменная. Обратите также внимание, что я описал операторы «меньше» и «больше» одновременно. Это не потому, что мне захотелось реализовать их оба, а потому, что так нужно. Если вы реализуете оператор «меньше», то обязательно нужно реализовать и оператор «больше». Это правило действует и наоборот. То есть реализовывать их можно только оба одновременно.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter5\MyPointProject2* сопровождающего книгу электронного архива (см. приложение).

5.7.3. Операторы преобразования

С точки зрения операторов мы сейчас будем рассматривать, наверное, самую интересную тему — преобразование несовместимых типов. Тема сама весьма интересна, но в то же время такая процедура может сделать ваш код некрасивым и неинтуитивным, — впрочем, это только на мой взгляд. С другой стороны, мы с этим получаем великолепную мощь и удобство программирования, что я вам сейчас и продемонстрирую.

Допустим, у нашего класса `Person` появились два свойства: `X` и `Y`, с помощью которых мы можем задавать координаты человека на карте города:

```
public int X { get; set; }
public int Y { get; set; }
```

А что если мы хотим получить эти координаты в виде класса точки `MyPoint`? Нужно создать новый объект класса точки и присвоить ему координаты человека из объекта `Person`. В случае с точкой это делается не так уж и сложно, ну а если классы содержат множество свойств? Копирование будет осуществляться гораздо сложнее, и если у вас в коде есть 20 вызовов такого преобразования, то наглядность потеряется.

Допустим, мы написали 20 вызовов преобразования следующего вида:

```
MyPoint point = new MyPoint(person.X, person.Y);
```

А что если нужно добавить в класс `Person` и в класс `MyPoint` новое свойство `PointName`, где будет храниться название здания или населенного пункта, в котором расположена точка? На мой взгляд, это катастрофа, потому что придется просмотреть все 20 преобразований в нашем коде и подкорректировать их так, чтобы они копировали из объекта `Person` еще и имя точки `PointName`. А если таких мест в коде будет не 20, а 500? Это вполне реальное число для большого проекта.

Намного лучше было бы, если бы преобразование объекта `Person` в `MyPoint` выглядело так:

```
MyPoint point = (MyPoint)person;
```

Однако по умолчанию это невозможно, и компилятор выдаст ошибку: **Cannot convert type Person to MyPoint** («Не могу конвертировать Person в MyPoint»), — но это только по умолчанию. Мы можем создать такой оператор преобразования, который научит компилятор и среду выполнения конвертировать даже несовместимые типы. В нашем случае мы должны в классе `MyPoint` написать следующий код:

```
public static explicit operator MyPoint(Person p)
{
    return new MyPoint(p.X, p.Y);
}
```

Это объявление очень сильно похоже на уже знакомые нам операторы, только здесь не возвращаются никакие значения, а вместо этого стоит ключевое слово `explicit`. Оно как раз и указывает на то, что перед нами оператор, который определяет преобразование типов. Вместо имени метода стоит имя текущего класса — можно еще сказать, что это класс, в который будет происходить преобразование. В скобках указывается класс, из которого выполняется преобразование.

В теле оператора мы должны создать новый экземпляр класса точки и скопировать в него нужные нам свойства. Так как нам требуются только координаты, то достаточно их и передать конструктору. Вот теперь преобразование типов пройдет без проблем, и следующая строка не вызовет никаких претензий со стороны компилятора:

```
MyPoint point = (MyPoint)person;
```

Теперь рассмотрим ситуацию с появлением нового свойства `PointName`. Нам все равно, сколько раз и где используется явное преобразование, — достаточно только подправить оператор в классе `MyClass`, и все будет работать:

```
public static explicit operator MyPoint(Person p)
{
    MyPoint point = new MyPoint(p.X, p.Y);
    point.PointName = p.PointName;
    return point;
}
```

Я надеюсь, что убедил вас в мощности такого подхода. Самое сложное в нем — определить те места, где нужно писать оператор преобразования, а где достаточно просто создать объект и скопировать в него нужные свойства без написания оператора явного преобразования. Я бы рекомендовал не писать оператор, если он будет использоваться только в одном-двух местах. Это лично моя рекомендация, а как поступите вы, решать уже вам самим.

Преобразование можно сделать еще и неявным. Если оператор описан как `explicit`, то в коде преобразования вы обязаны явно указывать в скобках перед переменной тип, в который происходит преобразование. Но если заменить ключевое слово `explicit` на `implicit`, то и этого делать не придется. Попробуйте сейчас поменять оператор на `implicit` и написать следующую строку кода:

```
MyPoint point = person;
```

Здесь мы даже в скобках не указываем тип, в который производится преобразование. Такой метод я рекомендую использовать еще реже и только в крайних случаях, потому что наглядность кода теряется еще сильнее. Тот, кто будет читать эту строку, должен будет держать преобразование в голове. Да и вы тоже через полгода не вспомните, где и какие операторы преобразования написали.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter5\ConversationProject* сопровождающего книгу электронного архива (см. приложение).

5.8. Шаблоны

Шаблоны — очень мощное средство, которое спасает нас от беспощадной необходимости писать килобайты одного и того же кода, когда нужно выполнить одни и те же операции над разными типами данных. Конечно, мы могли бы создать класс или метод, который работал бы с переменными `Object`, но это неудобно, потому что небезопасно. Шаблоны являются более мощным средством решения однотипных задач для разных типов данных, и наиболее явно эту мощь они проявляют в динамических массивах, которые мы будем рассматривать позже.

Для начала посмотрим, как можно создать шаблон простого метода:

```
static string sum<T>(T value1, T value2)
{
    return value1.ToString() + value2.ToString();
}
```

Я сделал метод статичным только потому, что в примере, приведенном в сопровождающем книгу электронном архиве, он вызывается в консольном приложении из статичного метода `Main()`. Это необязательный атрибут, и шаблонные методы могут быть и нестатичными.

Самое интересное записано сразу после имени метода в угловых скобках. Тут объявляется какая-то строка или буква, которая является шаблоном. Еще с классов языка C++ пошло негласное правило именовать шаблоны буквой `T` (от слова *Template* — шаблон). Вы можете назвать шаблон по-другому, но я рекомендую придерживаться этого правила. В коде буква `T` будет заменяться на тип данных, который вы станете передавать при использовании метода или класса.

В скобках мы указываем, что метод принимает два параметра какого-то типа `T`, который определится на этапе вызова метода. В теле метода каждая переменная приводится к строке и выполняется конкатенация строк.

Теперь посмотрим, как этот метод может использоваться:

```
static void Main(string[] args)
{
    string intsum = sum<int>(10, 20);
    Console.WriteLine(intsum);
}
```

```
string strsum = sum<string>("Hello, ", "world");  
Console.WriteLine(strsum);  
  
Console.ReadLine();  
}
```

Сначала мы вызываем метод `sum<int>`. В угловых скобках указан тип `int`, а значит, в нашем шаблоне везде, где была буква `T`, станет подразумеваться тип данных `int`, т. е. метод будет принимать числа. Именно числа мы и передаем этому методу.

Второй раз мы вызываем этот метод, указывая в качестве шаблона строку `sum<string>`. На этот раз параметры передаются как строки. Если попытаться передать числа, то произойдет ошибка. Именно это и определяет мощь шаблонов. Если метод объявлен с шаблоном определенного типа, то только параметры этого типа могут передаваться методу. За этим следит компилятор, и он не даст использовать разнотипные данные.

Внутри шаблонного метода по умолчанию переменная будет иметь тип `Object`, поэтому мы ограничены только методами, которые есть у этого класса. В приведенном примере ничто не указывает на тип `T`, но на самом деле можно было бы написать этот код так:

```
static string sum<T>(T value1, T value2) where T : Object  
{  
    return value1.ToString() + value2.ToString();  
}
```

Здесь я явно говорю, что `T` является объектом. Так как в `C#` все классы в какой-то момент наследуют `Object`, то `where T : Object` можно опускать и не указывать.

Если вы создаете метод, который будет работать с определенным классом или его потомками, то это можно сделать, добавив имя класса после объявления метода:

```
static string DisplayName<T>(T p) where T : Person  
{  
    return p.FirstName + p.LastName;  
}
```

Здесь после объявления метода и до начала кода стоит фильтр `where T : Person`, который говорит, что `T` может быть классом `Person` или любым его потомком.

В качестве типа `T` можно указывать интерфейсы, но мы их еще не проходили, а именно с ними шаблоны становятся невероятно гибкими.

На методах показать мощь шаблонов сложно, и я не смог придумать более интересного примера. Зато на классах шаблоны проявляют себя намного лучше. Давайте попробуем написать класс поддержки массива из 10 значений. Это не динамический, а статический массив — для простоты примера, но за счет шаблонов его использование становится универсальным.

Итак, если вам нужен массив небольшого размера, то его можно реализовать в виде статического массива, а для удобства написать вспомогательный класс, код которого приведен в листинге 5.2.

Листинг 5.2. Вспомогательный класс для работы с массивом

```
public class TemplateTest<T>
{
    T[] array = new T[10];
    int index = 0;

    public bool Add(T value)
    {
        if (index >= 10)
            return false;

        array[index++] = value;
        return true;
    }

    public T Get(int index)
    {
        if (index < this.index && index >= 0 )
            return array[index];
        else
            return default(T);
    }

    public int Count()
    {
        return index;
    }
}
```

При объявлении класса, использующего шаблон, буква шаблона указывается в угловых скобках после имени класса. Теперь внутри класса вы можете работать с буквой `T` как с типом данных — объявлять переменные, получать параметры в методе и даже возвращать значения типа `T`. Создавая конкретный объект класса `TemplateTest`, во время объявления вы должны будете указать, для какого типа его создаете. Например, объявление `TemplateTest<int> testarray` заставит систему создать массив `testarray` для хранения чисел, и все методы, где использовалась буква `T`, будут работать с числами. Получается, что при использовании шаблона класса вы как бы с помощью замены вставляете везде вместо `T` указанный тип.

Вот так этот шаблон может быть использован для массива чисел:

```
TemplateTest<int> testarray = new TemplateTest<int>();

testarray.Add(10);
testarray.Add(1);
testarray.Add(3);
testarray.Add(14);
```

```
for (int i = 0; i < testarray.Count(); i++)  
    Console.WriteLine(testarray.Get(i));
```

Для того чтобы создать точно такой же массив, но для хранения строк, достаточно только изменить объявление:

```
TemplateTest<string> testarray = new TemplateTest<string>();
```

Нам не нужно писать новый класс, который будет реализовывать те же функции для типа данных `string`, — благодаря шаблону все уже готово.

В случае с классом мы также можем после слова объявления класса добавить `where` и указать, чем является этот тип. Уточнить, чем является `T` в случае с объявлением класса, можно следующим образом:

```
public class TemplateTest<T> where T: XXXXX
```

Здесь `XXXXX` может принимать одно из следующих значений:

- ☐ `class` — шаблон может быть создан для классов. Причем класс не должен быть объявлен как `sealed`, иначе его использование не имеет смысла;
- ☐ `struct` — шаблон может быть создан на основе структур, т. е. в нашем случае буква `T` может заменяться только на структуру;
- ☐ `new()` — параметр `<T>` должен быть классом, имеющим конструктор по умолчанию;
- ☐ *Имя_класса* — шаблон может быть создан только для типов данных, являющихся наследниками указанного класса;
- ☐ *Имя_интерфейса* — шаблон может быть создан только для классов, реализующих указанный интерфейс.

Следующий пример объявляет шаблон для хранения классов, являющихся наследниками `Person`:

```
public class TemplateTest<T> where T : Person
```

Вы не сможете создать теперь массив чисел на основе этого шаблона — только массив людей. Такое ограничение очень удобно, если вам действительно нужно хранить данные определенного класса. В этом случае вы можете безболезненно приводить переменную `T` к этому базовому классу, потому что мы точно знаем, что массив может быть создан лишь для хранения наследников `Person`.

ПРИМЕЧАНИЕ

Исходный код примеров к этому разделу можно найти в папках `Source\Chapter5\TemplateProject1` и `Source\Chapter5\TemplateProject2` сопровождающего книгу электронного архива (см. приложение).

5.9. Анонимные типы

Отличие *анонимного типа* от прочих заключается в том, что у него нет имени. Когда мы создаем экземпляр какого-либо класса, то указываем тип класса и после этого можем задать значения свойств:


```
Person p = new Person() {  
    FirstName = "Михаил",  
    LastName = "Фленов",  
    Age = 10  
};
```

Чтобы этот код сработал, где-то должен быть объявлен класс `Person` и объявлены все его свойства.

Далеко не всегда бывает удобно писать объявление классов для каждого возможного случая. Если просто нужно создать какой-то единичный объект из хорошо структурированных данных, то можно использовать анонимный тип:

```
var p = new {  
    FirstName = "Михаил",  
    LastName = "Фленов",  
    Age = 10  
};
```

Здесь мы не указываем никакого класса, потому что его нет — он анонимный или, если по-другому сказать, без имени. В остальном задание свойств выглядит точно так же, как и в предыдущем примере.

Так как тип данных анонимный, то какого типа объявлять переменную? Тут есть два варианта: использовать `var`, как я сделал в этом примере, или объявлять переменную как `Object`, потому что все классы в .NET происходят от этого класса, даже если мы его не указываем. Первый вариант с `var` — предпочтительнее. Я говорил, что предпочитаю указывать всегда типы данных явно и не использовать `var`, но в нашем случае именно `var` подходит идеально, и отчасти ради этого его и создавали когда-то.

Как я узнал, что у анонимного типа будет свойство `FirstName` и что я могу присвоить ему значение? Ведь нигде нет описания класса, и мы не знаем, как он выглядит. На самом деле никакой ошибки тут нет — вы можете в фигурных скобках присваивать значения любым именам свойств, и компилятор создаст анонимный класс с теми свойствами, которым вы присвоите значения во время инициализации. Тип этих свойств будет определен в зависимости от контекста, как это происходит в случае с `var`.

В нашем примере будет создан класс без имени, обладающий тремя свойствами: `FirstName`, `LastName` и `Age`. Первые два будут строками, а последнее — числом.

Все свойства устанавливаются только для чтения, т. е. попытка изменения приведет к ошибке уже на этапе компиляции:

```
p.FirstName = "Миша"; // Ошибка
```

Наверное, самое яркое преимущество использования анонимных типов данных — LINQ (Language Integrated Query, интегрированные в язык запросы), о чем мы поговорим в *главе 10*.

5.10. Кортежи

Кортежи (от *англ.* Tuple) используются для описания структуры данных, состоящей из последовательности элементов.

На мой взгляд, это не самая важная структура данных, и может быть поэтому она появилась в составе платформы .NET совсем недавно. Нет ничего такого, что может делать кортеж, чего нельзя реализовать с помощью классов или структур.

Но кортежи все же удобны тем, что позволяют сгруппировать данные в одну переменную без необходимости объявлять новый класс. Если метод должен вернуть два значения, то можно использовать как раз кортежи. Хотя я чаще для этого объявляю отдельную структуру и класс.

Допустим, метод должен подсчитывать значения координат *X* и *Y*. Как вернуть оба значения сразу? Можно объявить класс из двух свойств, можно объявить структуру, а можно ничего не объявлять и использовать кортежи (Tuple), как показано в листинге 5.3.

Листинг 5.3. Пример использования кортежей

```
static void Main(string[] args) {  
    Tuple<int, int> coordinates = new Tuple<int, int>(10, 20);  
    Console.WriteLine(coordinates.Item1 + " " + coordinates.Item2);  
  
    var coordinates2 = GetCoordinates();  
    Console.WriteLine(coordinates2.Item1 + " " + coordinates2.Item2 + " " +  
        coordinates2.Item3 + " " + coordinates2.Item4);  
    Console.ReadLine();  
}  
  
static Tuple<int, int, int, string> GetCoordinates() {  
    return new Tuple<int, int, int, string>(10, 20, 30, "Что-то");  
}
```

В первой строке метода `Main` создается новый объект `Tuple`. В угловых скобках через запятую нужно указать тип данных для каждого элемента структуры данных, а их может быть до 8 штук. Но в нашем случае всего два элемента, и оба числовые.

После знака равенства мы инициализируем новый кортеж примерно так же, как это происходит с инициализацией классов. Доступ к элементам кортежа осуществляется через свойства `ItemX`, где *X* — это порядковый номер элемента в том же порядке, как мы их и создавали. Нумерация идет от 1 до 8.

А под этим кортежем я создал еще один метод: `GetCoordinates()`, который не делает ничего полезного, а только возвращает `Tuple` из четырех элементов, три из которых числа и еще один — строка.

5.11. Форматирование строк

При выводе в консоль я очень часто использую простой подход с объединением строк. Допустим, у нас есть переменная `index`, которую нужно вывести в консоль. С помощью объединения строк это можно сделать так:

```
int index = 10;
Console.WriteLine("Index variable = " + index);
```

Это достаточно просто и прекрасно работает, но начинает выглядеть некрасиво, когда у вас много переменных должны объединяться в специально сформатированную строку или когда формат находится в файле ресурсов и должен быть локализован. Дело в том, что при локализации переменные могут попадать в разные места: где-то первым идет имя, а где-то — фамилия, и эти правила должны находиться в строке формата, а нам надо только предоставить переменные.

Это можно решить с помощью форматирования, когда внутри строки находятся специальные *токены*, которые будут заменяться на реальные значения. Такие токены оформляются в виде чисел в фигурных скобках:

```
Console.WriteLine("Index variable = {0}", index);
```

Имеющийся в этой строке токен `{0}` будет во время выполнения заменен на значение нулевой переменной. Переменных может быть несколько:

```
Console.WriteLine("Вас зовут {0} {1}", FirstName, LastName);
```

В этой строке имеются два токена — с номерами 0 и 1, и во время выполнения токен `{0}` будет заменен на значение `FirstName`, а токен `{1}` — на значение `LastName`. Число здесь — это порядковый номер переменной.

Все сказанное касается вывода в консоль, а если нам не нужно выводить в консоль? В этом случае можно использовать метод `String.Format`, который заменяет токены и возвращает результат:

```
String.Format("Index variable = {0}", index);
```

Вычислять токены по номерам не всегда удобно, особенно если строка большая. В этом случае намного эффективнее было бы вместо номеров в фигурных скобках указывать сразу же имя переменной. И это возможно, если перед началом строки поставить символ доллара:

```
 $"Index variable = {index}"
```

Этот метод называется *интерполяцией строк*. Если перед строкой стоит символ доллара, то имена в фигурных скобках будут восприниматься как C#-код, и тут можно держать переменные или даже выполнять какие-то небольшие вычисления.

В следующем примере я не просто вывожу значение переменной `index` — сначала оно увеличивается на 5, и потом уже выводится результат:

```
 $"Index variable = {index + 5}"
```

5.12. Динамика

В C# ради безопасности лучше работать с определенными типами. Мы можем использовать базовый тип `object` для хранения любых объектов, но и в этом случае все равно происходит проверка типов и безопасности доступа уже на этапе компиляции.

Давайте создадим класс с одним лишь методом, который будет выводить что-то в консоль:

```
class Test
{
    public void Print()
    {
        Console.WriteLine("Test");
    }
}
```

Сможем ли мы выполнить теперь такой код?

```
object test = new Test();
test.Print();
```

Здесь в первой строке мы помещаем в переменную типа `object` экземпляр класса `Test`, а во второй строке я пытаюсь вызвать метод `Print`. Хотя такой метод действительно есть у класса `Test`, мы все же работаем с переменной класса `object`, поэтому уже на этапе компиляции произойдет ошибка.

Чтобы все же вызвать метод `Print`, нужно сделать приведение — например, так:

```
(test as Test).Print();
```

В редких случаях может понадобиться вызвать метод без приведения, и для этого мы можем отключить статическую проверку на этапе компиляции за счет использования типа `dynamic`:

```
dynamic test = new Test();
test.Print();
test.Start();
```

Здесь переменная `test` объявлена как `dynamic`, и поэтому во второй строке я смог вызвать метод `Print` без приведения типов. На этапе компиляции проверки не будет, а уже после запуска во время выполнения платформа будет искать у объекта `test` метод `Print` и с успехом найдет его.

Это все прекрасно, но проблема может возникнуть в третьей строке. В ней я вызываю метод `Start`, которого нет у класса `Test`. За счет динамики проверка на этапе компиляции отключена, а во время выполнения произойдет ошибка, потому что метод `Start` не будет найден.

Не используйте `dynamic` без особой надобности, потому что это может дать волю ошибкам, которые потом сложно будет отловить.

При этом за счет полиморфизма вы сможете вызвать методы с некорректными параметрами. Например:

```
dynamic test = new Test();  
test.Print("Hello");
```

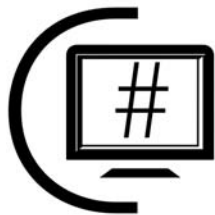
Такой код будет скомпилирован без проблем, но вот работать не будет, — ведь метод `Print` не принимает никаких параметров, а мы пытаемся это сделать, поэтому во время выполнения снова будет получена ошибка.

А что если мы присвоим результат какой-то операции переменной `var`:

```
dynamic s = "test";  
var sum = s + 3;
```

Здесь происходит сложение `dynamic` с числом, и результат сохраняется в переменной типа `var`. Мы знаем, что тип данных `var` фиксируется при первом использовании и потом его поменять нельзя. Какой тип будет у `sum`? Все очень просто, эта переменная будет динамичной `dynamic`.

ГЛАВА 6



Интерфейсы

Допустим, нам нужно сделать так, чтобы два класса могли иметь одинаковые методы. Если мы проектируем эти классы и они происходят от одного предка, который также разрабатывали мы, и если классы схожи по смыслу, то задачу можно решить легко — встроить объявление метода в класс-предок для обоих классов. Это объявление может быть как абстрактным, так и полноценным, с реализацией по умолчанию.

Но не кажется ли вам, что слишком много условий «если»? Мне кажется, что да, потому что все условия удается соблюдать далеко не всегда. Очень часто нарушается условие родственности классов, и нужно наделить два совершенно разных класса одним и тем же методом. Просто создать в классах методы с одним и тем же именем мало, необходимо еще и получить возможность использовать объектное программирование, а точнее — полиморфизм. Тут имеется в виду возможность вызывать метод вне зависимости от класса, т. е. не имеет значения, какой перед нами класс, важно то, чтобы у него был нужный нам метод или свойство с определенным именем.

Подобную проблему легко было решить в таком языке, как C++, благодаря *множественному наследованию* — когда один класс может наследоваться от нескольких и наследовать все их свойства и методы. Множественное наследование — мощная возможность, но очень опасная, потому что приводит к проблемам, когда классу нужно наследовать два других класса, у которых есть свойство или метод с одним и тем же именем. Как потом разделять эти методы и свойства? Как потом их использовать?

В современных языках множественное наследование отсутствует, вместо него введено понятие *интерфейсов*, которые и решают такую задачу намного лучше.

Когда я учился программированию, то сразу не смог понять, в чем прелесть интерфейсов, потому что не было хорошего примера, который бы раскрыл мне их преимущества. И по своему опыту могу сказать, что очень часто люди сталкиваются с подобными проблемами непонимания сути этой темы.

6.1. Объявление интерфейсов

Интерфейс — это объявление, схожее с классом, но в нем нет реализаций методов, т. е. все методы его абстрактны и не содержат реализации. С их помощью вы можете описать, какими функциями должен обладать класс и что он должен уметь делать. При этом интерфейс не имеет самого кода и не реализует функции. И это было отличным объяснением до появления C# 8-й версии, где Microsoft добавила возможность иметь реализацию по умолчанию. Так что сначала мы разберемся с интерфейсами, притворившись, что нельзя делать реализации, потому что именно так они задумывались изначально, а потом уже рассмотрим нововведение C# 8 (см. *разд. 6.8*).

Объявление интерфейса начинается со слова `interface`. В отличие от классов, интерфейсы не наследуют никакого класса, даже `Object`, автоматически наследуемого для всех классов. Вы также можете не указывать модификаторы доступа для описываемых методов — все они считаются открытыми. В этом весь смысл интерфейса: описать действия, через которые мы потом сможем взаимодействовать с классами, которые реализуют интерфейс.

В *главе 3* мы написали небольшой класс `Person`. Давайте его вспомним и на его основе покажем пример использования интерфейса. Чем таким можно наделить человека, чтобы понадобился интерфейс? Я долго думал и решил наделить его функциями хранения денег. Деньги могут храниться не только в кармане у человека, но и в кошельке или в сейфе. Все это разные по идеологии классы объектов, поэтому пытаться реализовать их из одного класса будет самоубийством, — намного эффективнее описать функции, которые нужны для работы с деньгами в интерфейсе, и реализовать интерфейс в классе.

Следующий пример показывает, как можно создать интерфейс кошелька:

```
interface IPurse
{
    int Sum
    {
        get;
        set;
    }

    void AddMoney(int sum);
    public int DecMoney(int sum);
}
```

Выглядит все прям как объявление класса с абстрактными методами, но вот с точки зрения реализации это не одно и то же. Созданный нами интерфейс имеет два метода: у первого нет модификаторов доступа, но будет подразумеваться `public`, у второго — явно указан модификатор `public`. А можно указать `private`? Да, но с определенными ограничениями, и об этом чуть позже.

Интерфейсы создаются для описания методов, которые были бы общедоступны, и чтобы эти методы вызывали другие классы, а значит, не имеет смысла описывать

закрытые методы и свойства, хотя это все же возможно в C#. Зато для самого интерфейса можно указать, является он открытым или нет.

Интерфейс не может иметь переменных, потому что открытые переменные — нарушение объектно-ориентированного подхода. Зато вы можете объявлять свойства. Вот это совсем не запрещено.

Обратите внимание на имя интерфейса — оно начинается с буквы **I** (от слова *Interface*). Такое именование не является обязательным, и вы можете именовать интерфейс как угодно. Но все же в .NET принято, чтобы имя интерфейса начиналось именно с буквы **I**, и я рекомендую вам придерживаться этого соглашения, потому что оно очень удобно. По одной букве можно сразу понять, что перед вами.

Впрочем, я встречал рекомендацию, что имена интерфейсов могут не начинаться с **I**, а включать фрагмент *Interface* в конце имени, и таким образом мы должны были бы назвать наш кошелек *PurseInterface*. Мне такой подход не по душе, потому что тогда имена интерфейсов в C# будут начинаться с **I** (этого мы уже изменить не можем), а наши интерфейсы — заканчиваться на слово *Interface*. Мое главное правило — какой бы метод именования ни существовал, главное, чтобы все было в одном стиле, и если в .NET уже есть какой-то стиль, то менять его не следует.

Теперь нужно определиться, где писать интерфейсы. В принципе, это такой же код, и вы можете добавить его описание в одном файле с классом, просто расположив их рядом в одном пространстве имен:

```
namespace InterfaceProject
{
    interface IPurse
    {
        // интерфейс
    }

    class SomeClass : IPurse
    {
        // класс
    }
}
```

Но лучше все же располагать каждый интерфейс в отдельном файле, как мы это делаем с классами. Для создания файла для интерфейса не требуется ничего сверхъестественного — достаточно щелкнуть правой кнопкой мыши на проекте и выбрать в контекстном меню **Add | New Item**. В открывшемся окне можно выбрать пункт **Class**, а можно и **Interface**. Оба пункта создают файл кода, разница только в том, какие пространства имен подключаются по умолчанию.

Поскольку интерфейс не имеет реализации у методов, а только объявляет их как абстрактные методы в классах, то вы *не можете* создать экземпляр интерфейса. Не для этого мы их объявляли.

Вы можете использовать шаблон создания нового класса — просто после создания файла измените в объявлении `class` на `interface`.

6.2. Реализация интерфейсов

Интерфейсы должны быть где-то реализованы, иначе они бесполезны. Реализация интерфейсов схожа с наследованием. Класс как бы «наследует» интерфейс и таким образом обещает реализовать все его методы. Один класс может реализовать более одного интерфейса. Если класс может быть наследником только одного класса, то интерфейсов он может реализовывать любое количество.

Давайте наделим нашего человека (класс `Person`) кошельком. Для этого реализуем интерфейс следующим образом:

```
class Person: IPurse
{
    // реализация класса
}
```

Так как любой класс, если он ничего не наследует явно, автоматически наследует класс `Object`, то предыдущая запись идентична следующему объявлению:

```
class Person: Object, IPurse
{
    // реализация класса
}
```

В этом случае мы явно говорим, что наш класс `Person` является наследником `Object` и реализует интерфейс `IPurse`. Если вы наследуете класс и интерфейс одновременно, то имя класса должно быть написано первым.

Но мало просто указать интерфейс среди наследников. Вы должны написать внутри своего класса реализацию всех свойств и методов интерфейса. Причем все они должны быть открытыми: `public`. Если вы забудете написать реализацию хотя бы одного метода или свойства, то компилятор выдаст ошибку компиляции. Интерфейсы должны реализовываться полностью или не реализовываться вовсе.

Возможный класс `Person` вместе с реализацией представлен в листинге 6.1.

Листинг 6.1. Реализация интерфейса `IPurse`

```
class Person: IPurse
{
    // здесь методы класса Person
    // ...
    // ...

    // Далее идет реализация интерфейса
    int sum = 0;
```

```
public int Sum
{
    get { return sum; }
    set { sum = value; }
}

public void AddMoney(int sum)
{
    Sum += sum;
}

public int DecMoney(int sum)
{
    Sum -= sum;
    return Sum;
}
}
```

Внутри класса есть методы `AddMoney()` и `DecMoney()`, а также свойство `Sum`, которые уже были объявлены в интерфейсе. Мы их реализовали так же, как реализовывали абстрактные классы. Тут очень важным является то, что реализовывать приходится не только методы, но и свойства. Все свойства, указанные в интерфейсе, должны быть прописаны и в классе, реализующем интерфейс.

6.3. Использование реализации интерфейса

Теперь посмотрим, как можно использовать интерфейсы на практике. Самый простой метод использования — создать класс и просто вызывать методы. Например:

```
Person person = new Person("Михаил", "Фленов");
person.AddMoney(1000000);
```

В этом примере создается экземпляр класса `Person` с моими именем и фамилией, после чего вызывается метод `AddMoney()`, чтобы положить в кошелек этого человека миллиончик. Так как «этим человеком» являюсь я сам, то себе в карман хотелось бы положить именно столько денег, — надеюсь, что долларов, потому что за столько рублей квартиру нигде не купить. Вернемся к вызову. Тут ничего сложного нет, потому что мы просто вызываем метод `AddMoney()`, который реализован в классе. Несмотря на то что его объявление было вынужденным из-за реализации интерфейса `IPurse`, метод остается методом, и мы можем вызывать его, как и любые другие методы класса.

Но в таком вызове не видно всей мощи интерфейса. Чтобы увидеть ее, нужно посмотреть на листинг 6.2, где я использую методы интерфейса различными способами, которые будут более интересны.

Листинг 6.2. Пример использования реализации интерфейса

```
Person person = new Person("Михаил", "Фленов");
Object personObject;
IPurse purse;

personObject = person;
purse = person;

if (personObject is IPurse)
{
    ((IPurse)personObject).AddMoney(100);
    Console.WriteLine("Сумма в кошельке: " +
        ((IPurse)personObject).Sum.ToString());
}

purse.DecMoney((int)50);
Console.WriteLine("Сумма в кошельке: " + purse.Sum.ToString());
```

Теперь самое интересное — посмотрим, что происходит в этом листинге. В самом начале объявляются три переменные:

- ❑ `person` — классическая переменная класса `Person`, которая сразу же инициализируется;
- ❑ `personObject` — переменная самого базового класса `Object`. Ей я присваиваю значение переменной `person`;
- ❑ `purse` — переменная типа интерфейса `IPurse`. Мы не можем инициализировать интерфейсные переменные интерфейсами, потому что в интерфейсе нет реализации. Зато мы можем объявлять интерфейсные переменные и присваивать им значения.

В качестве значения интерфейсной переменной можно присвоить объект любого класса, который реализует этот интерфейс. В нашем случае класс `Person` реализует `IPurse`, значит, переменной `purse` мы можем присвоить любой объект класса `Person`. Например, мы могли бы проинициализировать переменную так:

```
IPurse purse = new Person("Михаил", "Фленов");
```

Но в методе `Main` я присваиваю ей значение переменной `person`. Получается, что все три переменные (`person`, `personObject` и `purse`), несмотря на разный класс, имеют одно и то же значение, и мы сможем через них работать с одним и тем же объектом. Так как с объектной переменной мы можем работать без проблем, то в этом примере посмотрим, как можно работать через интерфейс.

После инициализации мы пробуем добавить в кошелек деньги. Раньше, чтобы вызвать метод потомка, мы приводили переменную, указывая перед ней имя реально-го класса. То есть мы должны были бы написать:

```
((Person)personObject).AddMoney(100);
```

Здесь же мы приводим переменную `personObject` к классу `Person` (каким она и является на самом деле) и вызываем его метод `AddMoney()`. Методу передается значение числа, введенного в компонент `numericUpDown1`, приведенное к числу `int`.

У этого способа есть недостаток, т. к. он привязывается к определенному классу. В нашем случае перед нами может быть любой класс, который реализует интерфейс кошелка, и не обязательно `Person`. Можно сделать проверку, каким классом является переменная, и приводить переменную к этому классу, а можно использовать интерфейс и приводить к нему:

```
(IPurse)personObject).AddMoney(100);
```

Да, вы можете приводить переменную к интерфейсу, и если объект, который находится в переменной, реализует этот интерфейс, то код выполнится корректно. Соответственно корректно выполнится и наш пример, поскольку в переменной `personObject` класса `Object` действительно находится объект класса `Person`, а этот класс реализует нужный нам интерфейс.

Прежде чем выполнять приведение, лучше убедиться, что объект реализует интерфейс, а это можно сделать с помощью ключевого слова `is`. Это ключевое слово может проверять принадлежность переменной не только к классу, но и объекта к интерфейсу. Если вы уверены, что в переменной находится класс, который реализует интерфейс, то проверку делать необязательно. Несмотря на то что я в нашем случае уверен, однако такую проверку добавил.

Приведение объектов можно делать двумя способами: указать нужный тип в скобках перед переменной или использовать ключевое слово `as`. Приведение типов к интерфейсу тоже можно делать двумя способами. Первый мы уже использовали, а второй будет выглядеть следующим образом:

```
(personObject as IPurse).AddMoney(100);
```

Здесь в первых скобках переменная `personObject` приводится к типу `IPurse`. Если это возможно и объект в переменной действительно реализует этот интерфейс, то такая операция завершится удачно. Если объект не реализует интерфейса, то операция завершится ошибкой.

По нажатию кнопки уменьшения денег мы напрямую вызываем методы интерфейса через интерфейсную переменную:

```
purse.DecMoney(50);
```

Таким образом, нам абсолютно все равно, объект какого класса находится в переменной, — главное, чтобы этот объект реализовывал интерфейс, и тогда его метод будет вызван корректно.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter6\InterfaceProject* сопровождающего книгу электронного архива (см. приложение).

6.4. Интерфейсы в качестве параметров

Интерфейсы удобны не только с точки зрения унификации доступа к методам, но и для получения универсального типа данных, который можно передавать как переменные в другие методы. Ярким примером передачи параметров типа интерфейсов являются коллекции, с которыми мы встретимся в *главе 7*.

Давайте сейчас напомним в нашем тестовом приложении метод, который будет универсально уменьшать деньги в кошельке:

```
void DecMoney(IPurse purse)
{
    purse.DecMoney(100);
    Console.WriteLine(purse.Sum.ToString());
}
```

В качестве параметра метод получает интерфейс. Это значит, что мы можем передать в метод любой класс, который реализует интерфейс, и метод корректно работает. Например, чтобы забрать деньги у нашего человека, мы могли бы вызвать этот метод следующим образом:

```
DecMoney(person);
```

Вы даже можете возвращать значения типа интерфейсов. Например, у вас может быть интерфейс `IMailClient`, который описывает методы отправки почты. Конкретными реализациями этого интерфейса могут быть `GmailClient`, `YandexClient` и т. д. Теперь можно создать метод, который в зависимости от ряда условий будет возвращать конкретную реализацию:

```
IMailClient GetEmailClient() {
    if (something) {
        return new GmailClient();
    }
    if (somethingelse) {
        return new YandexClient();
    }
}
```

Так мы можем создавать очень гибкий код, который будет отсылать сообщения с помощью различных почтовых клиентов. Использование этого метода может выглядеть так:

```
var emailClient = Factory.GetEmailClient();
emailClient.SendEmail("to@email.com", "Subject", "Body");
```

Это, конечно, гипотетический пример. Чтобы лучше разобраться с подобными решениями, я бы рекомендовал познакомиться с *паттернами программирования* (Design Patterns). На эту тему есть несколько книг, но здесь я не могу посоветовать ничего конкретного, потому что тех книг, которые я читал в свое время, уже давно не найти.

6.5. Перегрузка интерфейсных методов

А что будет, если в классе, который мы используем, уже есть метод с именем, который должен быть реализован из интерфейса? Причем параметры и все типы совпадают! Очень интересный вопрос, особенно если эти методы должны действовать по-разному. Получается, что мы должны в классе реализовать два метода с одинаковыми именами, параметрами и возвращаемыми значениями. Судя по правилам перегрузки методов, такой трюк не должен пройти, и компиляция подобного примера должна завершиться неудачей. Но в случае с методами интерфейсов такой способ имеет право на существование.

Допустим, у нашего класса `Person` есть метод, который добавляет в кошелек удвоенную сумму денег, переданную в качестве параметра:

```
public void AddMoney(int sum)
{
    Sum += sum * 2;
}
```

Теперь нам нужно рядом реализовать такой же метод, который будет являться реализацией `AddMoney()` из интерфейса `IPurse` и должен увеличивать содержимое кошелька на переданную в качестве параметра сумму без удвоения. Это можно сделать следующим образом:

```
void IPurse.AddMoney(int sum)
{
    Sum += sum;
}
```

Перед именем метода через точку написано имя интерфейса. Так мы явно указали, что именно этот метод является реализацией метода интерфейса `IPurse`.

Следует также обратить внимание, что пропал модификатор доступа `public`. В общем-то, он и раньше не был необходим, потому что все методы интерфейсов автоматически являются открытыми. А вот в нашем случае модификатор доступа указывать запрещено. Если вы явно указываете название интерфейса, который реализует метод, то указание модификатора доступа приведет к ошибке компиляции. Этот метод и так будет открытым.

А как теперь вызывать эти методы и различить их во время выполнения программы? Если у нас просто переменная класса, то будет вызван метод класса, который удваивает сумму:

```
Person person = new Person("Михаил", "Фленов");
person.AddMoney(10);
```

Чтобы вызвать метод интерфейса, мы должны явно привести переменную к интерфейсу, чем и укажем, что нам нужен метод именно интерфейса:

```
((IPurse)person).AddMoney(100);
```

Вот в этом примере будет вызван метод без удвоения.

Точно таким же образом мы можем решить проблему, когда класс наследует несколько интерфейсов, каждый из которых имеет метод с одним и тем же именем и одинаковыми параметрами, что может привести к конфликту. Например, у нас может быть интерфейс с именем `ITripplPurse` с такими же именами методов, но только они должны утраивать значение. Класс `Person` может наследовать сразу оба интерфейса без каких-либо конфликтов. Вариант такого решения показан в листинге 6.3.

Листинг 6.3. Наследование нескольких интерфейсов

```
class Person : IPurse, ITripplPurse
{
    // Методы класса Person
    ...

    // Реализация IPurse
    void IPurse.AddMoney(int sum)
    {
        Sum += sum;
    }

    int IPurse.DecMoney(int sum)
    {
        Sum -= sum;
        return Sum;
    }

    // Реализация ITripplPurse
    void ITripplPurse.AddMoney(int sum)
    {
        Sum += sum * 3;
    }

    int ITripplPurse.DecMoney(int sum)
    {
        Sum -= sum * 3;
        return Sum;
    }
}
```

Обратите внимание, что после объявления имени класса мы указываем через запятую два интерфейса. Соответственно наш новый класс `Person` будет наследовать сразу их оба. Если учесть, что он еще и автоматически наследует класс `Object`, то у нас получилось аж тройное наследование.

Внутри класса есть реализация всех методов обоих интерфейсов, и перед именами методов явно указано, из какого интерфейса взята та или иная реализация. Теперь, если вы хотите вызвать метод `AddMoney()` интерфейса `IPurse`, то должны привести переменную объекта к этому интерфейсу:

```
Person person = new Person("Михаил", "Фленов");  
(IPurse)person.AddMoney((int)numericUpDown1.Value);
```

Ну а если нужно вызвать метод `AddMoney()` интерфейса `ITripplPurse`, то это следует явно указать с помощью приведения типов:

```
((ITripplPurse)person).AddMoney((int)numericUpDown1.Value);
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter6\InterfaceProject2* сопровождающего книгу электронного архива (см. приложение).

6.6. Наследование

Интерфейсы тоже позволяют наследование, и оно работает точно так же, как у классов. Если интерфейс наследует какой-либо другой интерфейс, то он наследует все его методы и свойства.

Например, следующий интерфейс может являться описанием сейфа:

```
interface ISafe: IPurse  
{  
    bool Locked  
    {  
        get;  
    }  
  
    void Lock();  
    void Unlock();  
}
```

Этот интерфейс наследует методы кошелька, только в нашем случае они будут класть деньги не в кошелек, а в сейф, или забирать их оттуда. Помимо этого, интерфейс имеет методы открытия `Unlock()` и закрытия `Lock()` сейфа, а также булево значение `Locked`, которое будет возвращать значение `true`, когда сейф закрыт. Класс, который реализовывает наш сейф, должен быть внимателен и не позволять класть или вынимать деньги, когда дверца крепко закрыта. К тому же он должен реализовать все методы и свойства, которые мы описали в сейфе и которые были унаследованы от кошелька.

Таким образом, вы можете строить целые иерархии интерфейсов. Не знаю почему, но мне пока не приходилось использовать более двух уровней наследования. Может быть, не попадалось таких сложных задач. Но количество уровней наследования может быть любым и зависит от ваших предпочтений и надобности.

Интерфейсы поддерживают множественное наследование, но только интерфейсов. Классы наследовать нельзя! Интерфейсы вообще не могут наследовать классы.

6.7. Клонирование объектов

Есть такой фильм с Арнольдом Шварценеггером — «Шестой день», в котором сюжет построен на запрещении клонирования людей. В фильме это сделали потому, что человеческий мозг слишком сложен и просто так его клонировать не получалось. Неудачные попытки привели к тому, что пришлось такое клонирование запретить.

Классы в C# тоже могут быть сложными, но, несмотря на это, их клонирование не запрещено. Да, неправильное клонирование может привести к проблемам, но эта ответственность ложится на плечи программиста. Просто процесс клонирования должен быть контролируемым, и он таковым является.

Итак, что произойдет, если мы произведем простое присваивание:

```
Person p1 = new Person();  
Person p2 = p1;
```

Обе переменные — `p1` и `p2` — будут указывать на один и тот же объект в памяти. Изменяя свойство через переменную `p2`, мы изменяем объект, который инициализировался в `p1`.

А что если нужно создать именно копию объекта, чтобы в памяти появился еще один объект с такими же свойствами, но это должен быть уже другой объект, независимый от `p1`. Для этого можно поступить так:

```
Person p2 = new Person(p1.FirstName, p1.LastName);
```

Но это же не универсально! Для каждого метода нужно писать собственный код создания копии. А что если мы добавим к классу `Person` новое свойство? В этом случае придется найти все места, где мы клонировали код простым созданием нового объекта из конструктора, и изменить его.

А есть способ лучше? Конечно есть — реализовать интерфейс `ICloneable`. Интерфейс объявляет всего один метод, который вы должны реализовать в своем классе `Clone`. Задача этого метода — вернуть копию текущего объекта. В случае с нашим человеком это может выглядеть следующим образом:

```
public class Person: ICloneable  
{  
    ...  
    ...  
  
    public override Clone()  
    {  
        Person p = new Person(this.FirstName, this.LastName);  
  
        // здесь может быть перенос других свойств,  
        // которые не передаются через конструктор  
        // p.Свойство = this.Свойство
```

```
        return p;  
    }  
  
    ...  
    ...  
}
```

Метод создает свою копию и возвращает ее в виде универсального класса `Object`. Чтобы воспользоваться этим методом клонирования, мы должны всего лишь реализовать следующий код:

```
Person p1 = new Person();  
Person p2 = (Person)p1.Clone();
```

В этом примере для получения копии объекта вызывается метод `Clone()`. Копия возвращается в виде класса `Object`, а мы должны ее всего лишь привести к типу `Person`.

Теперь если в `Person` добавится новое свойство, то мы не должны бегать по всему проекту и искать ручное клонирование. Вместо этого нужно всего лишь подправить метод `Clone()`.

6.8. Реализация по умолчанию

Как уже отмечалось ранее, в C# 8 появилась возможность сделать реализацию методов интерфейса по умолчанию. Потом классы, которые будут реализовывать интерфейс, могут предоставить свою реализацию, а могут и оставить вариант по умолчанию.

У следующего интерфейса только один метод `Display`, и он отображает какую-то строку в консоли:

```
interface DefaultInterface  
{  
    public void Display()  
    {  
        Console.WriteLine("Test");  
    }  
}
```

Несмотря на то что интерфейс уже имеет реализацию, мы все еще не можем инициализировать этот тип, и следующая строка завершится ошибкой:

```
DefaultInterface interfaceTest = new DefaultInterface();
```

Вы должны обязательно создать класс, который будет реализовывать интерфейс:

```
class DefaultInterfaceImplementation: DefaultInterface  
{  
  
}
```

Созданный класс совершенно пустой и не реализует ни одного метода, и этого достаточно, потому что у единственного метода интерфейса есть реализация в интерфейсе. Вот этот класс мы уже можем инициализировать:

```
DefaultInterface test = new DefaultInterfaceImplementation();
```

Чтобы создать собственную реализацию метода интерфейса, просто добавляем ее к классу, как будто и не было никакой версии по умолчанию:

```
class DefaultInterfaceImplementation: DefaultInterface
{
    public void Display()
    {
        Console.WriteLine("New version");
    }
}
```

Я не люблю эту функциональность и пока еще ни разу не использовал ее в своих проектах или на работе. Предпочитаю относиться к интерфейсам как к контрактам, которые должны лишь определять доступные действия и не должны содержать кода. Считаю, что реализация по умолчанию имеет право на жизнь, но стоит добавлять ее только в крайнем случае.

С появлением реализации по умолчанию появилась возможность создавать частные (private) методы:

```
interface DefaultInterface
{
    public void Display()
    {
        Console.WriteLine("Test");
        PrivateMethod();
    }
    private void PrivateMethod() {
        Console.WriteLine("Приветный");
    }
}
```

Если у интерфейса есть private метод, то его реализацию нельзя будет предоставить потом в классе и нельзя будет вызывать из класса, но можно будет вызывать из другой реализации по умолчанию. Именно это и происходит в приведенном примере.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter6\InterfaceDefault* сопровождающего книгу электронного архива (см. приложение).

В случае с кошельком мы могли бы реализовать его методы прямо в интерфейсе:

```
interface IPurse
{
    int Sum
```

```
{
    get;
    set;
}

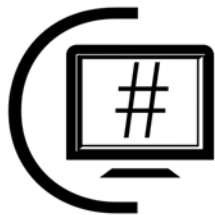
public void AddMoney(int sum)
{
    Sum += sum;
}

public int DecMoney(int sum)
{
    Sum -= sum;
    return Sum;
}
}
```

И теоретически это позволит нам иметь только одну реализацию в интерфейсе вне зависимости от того, какой класс перед нами: человек с кошельком, сейф, кассовый аппарат или что-либо другое. Но в этом случае мы получаем проблему наследования в ООП. Простое изменение поведения по умолчанию изменит поведение всех классов, которые используют его, и это может оказаться не совсем ожидаемым эффектом.

Если реализация действительно общая для всех, то тут можно решить проблему немного иным способом, но это уже вопрос не языка C#, а паттернов программирования и совершенно другой книги.

ГЛАВА 7



Массивы, списки, коллекции и словари

В этой главе мы подробнее разберемся с массивами, списками, коллекциями и словарями. Мы уже бегло познакомились с этими темами, но глубоко заглянуть в них не могли, поскольку не знали, что такое классы и интерфейсы. А знание интерфейсов для эффективной работы с массивами, списками, коллекциями и словарями просто необходимо, т. к. интерфейсы тут очень часто используются.

7.1. Базовый класс для массивов

В C# все типы данных являются классами, и массивы тоже. Когда вы объявляете массив, например `int[]`, то он автоматически наследуется от класса `Array` из пространства имен `System`. Класс `Array` предоставляет нам несколько статических методов:

- ❑ `BinarySearch()` — бинарный поиск, который позволяет получить достаточно быстрый результат на заранее отсортированном массиве. Чтобы воспользоваться методом, нужно реализовать интерфейс `IComparer`;
- ❑ `Clear()` — этот метод предоставляет возможность удалить из массива определенное количество символов. Например, чтобы удалить 5 символов, начиная со 2-го, в массиве `myArray` нужно выполнить:

```
Array.Clear(myArray, 2, 5);
```

- ❑ `CopyTo()` — позволяет скопировать данные одного массива в другой;
- ❑ `IndexOf()` — определяет индекс элемента в массиве. Если объект в массиве не найден, то метод вернет индекс наименьшего элемента в массиве минус единица. Индексы нумеруются с нуля, поэтому результатом вы увидите `-1`;
- ❑ `LastIndexOf()` — находит последний индекс объекта в массиве, если в массив несколько раз добавлен указанный объект;
- ❑ `Reverse()` — переворачивает массив в обратном направлении;
- ❑ `Sort()` — сортирует массив. Если в массиве находятся данные не простого типа, а классы, объявленные вами, то для сортировки нужно реализовать интерфейс `IComparer`. Для простых типов данных сравнение произойдет автоматически.

Класс также содержит свойство `Length`, которое позволяет определить количество элементов массива.

Примеры использования сортировки и разворачивания массива наоборот (реверс) можно увидеть в листинге 7.1.

Листинг 7.1. Использование статичных методов класса `Array`

```
int[] test = {10, 20, 1, 6, 15 };

// сортировка
Console.WriteLine("Отсортированная версия: ");
Array.Sort(test);
foreach (int i in test)
    Console.WriteLine(i);

// реверс элементов массива
Console.WriteLine("Реверсная версия: ");
Array.Reverse(test);
foreach (int i in test)
    Console.WriteLine(i);

// Удаление двух элементов массива
Console.WriteLine("Текущий размер: {0}", test.Length);
Array.Clear(test, 2, 2);
Console.WriteLine("После очистки: {0}", test.Length);

foreach (int i in test)
    Console.WriteLine(i);
```

Первые два блока кода просты и логичны. Сначала мы сортируем массив с помощью метода `Sort` и выводим его в консоль, а потом разворачиваем его на 180° и снова выводим на экран. Результат в консоли вполне понятен — сначала мы получаем отсортированный по возрастанию массив, а потом отсортированный массив разворачивается, т. е. остается отсортированным, но только по убыванию.

Самое интересное происходит в статичном методе очистки. Ему передаются три параметра: массив, из которого нужно очистить несколько элементов, индекс элемента, начиная с которого будут очищаться элементы, и количество элементов. Если сейчас запустить пример и посмотреть на размер массива до и после очистки двух элементов с помощью метода `Clear()`, то вы увидите, что размер массива никак не изменился. Но почему? Потому что `Clear()` не удаляет элементы, а *очищает*, а это большая разница. Элементы остаются на месте, просто становятся пустыми, а в случае с числовым массивом превращаются в ноль.

Класс `Array` и массивы, которые мы объявляем с помощью квадратных скобок `[]`, представляют собой классические массивы. В программировании можно встретить

разные понятия: массивы, списки и коллекции. Они очень схожи друг с другом, но есть все же и различия:

- ❑ *массивы* с давних времен — это последовательность значений фиксированной длины. Нельзя просто так взять и изменить размер массива — нужно выделять новую память и копировать данные в нее. Поэтому у массива нет методов добавления элементов. Нельзя просто так и удалить элемент, поэтому `Clear` не удаляет данные, и нет метода, который бы удалил определенный элемент;
- ❑ *списки* — это последовательности динамического размера, в которые можно добавлять элементы по мере необходимости и удалять;
- ❑ в *C#* существует несколько различных классов, которые позволяют работать с динамическими последовательностями, и их объединили одним термином — *коллекции*. Мы их рассмотрим в разд. 7.8.

7.2. Невыровненные массивы

До сих пор мы работали только с ровными многомерными массивами. Что это значит? Если мы объявляем двумерный массив в виде таблицы, то в каждой строке такого массива ровно столько элементов, сколько указано при создании, и это количество неизменно. А что если нам нужно создать двумерный массив, в котором в первой строке 1 элемент, во второй строке 10 элементов, в третьей 5 и т. д., и абсолютно бессистемно. Такие массивы называются *невыровненными* (*jagged*).

Невыровненный двумерный массив объявляется в виде:

```
Тип_данных[ ][ ] переменная;
```

А как его инициализировать, если каждая строка может иметь разное количество элементов? Да очень просто — вы должны проинициализировать массив только количеством строк, а вот количество колонок для каждой строки нужно указывать в отдельности. Например:

```
int[ ][ ] jaggedArray = new int[10][ ];  
jaggedArray[1] = new int[5];  
jaggedArray[2] = new int[2];  
jaggedArray[4] = new int[20];
```

В этом примере объявлена переменная `jaggedArray`, состоящая из 10 строк. После этого для отдельных строк задается явно количество элементов в строке. Количество элементов задано только для строк с индексами 1, 2 и 4. Остальные останутся нулевыми (непроинициализированными), и доступ к ним будет запрещен. При попытке прочитать или изменить значение за пределами проинициализированных элементов произойдет ошибка.

А как обращаться к элементам такого массива? Каждую размерность нужно указывать в отдельной паре квадратных скобок. Следующий пример изменяет элемент в строке 1 и колонке 2 на единицу:

```
jaggedArray[1][2] = 1;
```

Чуть более интересный пример приведен в листинге 7.2. В нем создается массив из 10 строк. Количество элементов в строке с каждой новой строкой увеличивается на единицу, создавая треугольный, а не прямоугольный массив элементов. Результат работы этого консольного примера показан на рис. 7.1.

Листинг 7.2. Невыровненный массив

```
int[][] jaggedArray = new int[10][];

// массив выделения памяти для каждой строки
for (int i = 0; i < jaggedArray.Length; i++)
    jaggedArray[i] = new int[i];

// использование массива
for (int i = 0; i < jaggedArray.Length; i++)
{
    for (int j = 0; j < jaggedArray[i].Length; j++)
    {
        jaggedArray[i][j] = j;
        Console.Write(jaggedArray[i][j]);
    }
    Console.WriteLine();
}
```

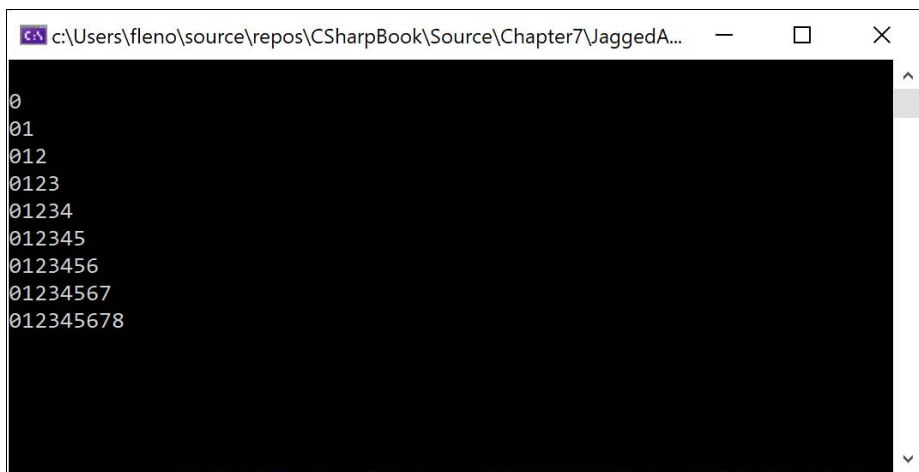


Рис. 7.1. Массив в виде треугольника

Обратите внимание, что количество элементов в массиве — 10, а мы видим треугольный результат в виде 9 элементов в строку и в колонку (числа от нуля до восьми), а не 10. Куда делся еще один элемент? Попробуйте сейчас увидеть ответ в коде примера.

Чтобы создать треугольный массив, после объявления невыровненного массива запускается цикл, который выполняется от нуля до количества элементов в массиве

ве. Чтобы определить количество элементов, мы используем свойство `Length`, о котором говорилось в *разд. 7.1*. Это свойство унаследовано от базового класса массивов.

Внутри цикла мы инициализируем очередную строку текущим значением счетчика в переменной `i`. Вот тут и скрыт ответ на вопрос, почему мы видим треугольный массив из 9 элементов, а не из 10. Дело в том, что на самом первом шаге счетчик равен нулю, а значит, в нулевой строке будет ноль элементов. Видите на *рис. 7.1* в самом верху пустую строку? С ее учетом у нас и получается полный комплект в 10 элементов.

Потом запускается еще один цикл, который перебирает все строки треугольного цикла, а внутри этого цикла запускается еще один цикл, который перебирает все элементы колонок текущей строки. Во внутреннем цикле значение элемента изменяется и тут же выводится в консоль.

7.3. Динамические массивы

Все массивы, которые мы создавали до этого момента, имели один очень серьезный недостаток — их размер был фиксированным и определялся во время инициализации. Но в реальной жизни далеко не всегда мы знаем, какого размера должен быть массив во время выполнения программы. Можно попытаться предсказать максимально возможный результат и выделить для его хранения максимально возможный размер, но при этом мы потратим слишком много памяти при отсутствии гарантии, что учли действительно самый страшный вариант.

Например, мы пишем программу, которая хранит количество машин на автостоянке. Какой величины массив создать? Допустим, что стоянка способна вместить 40 машин, и мы решили выделить с учетом запаса на всякий случай массив в 50 машин. А что если на стоянке останется стоять только одна машина? Память для 49 отсутствующих машин будет расходоваться без толку, и оставшейся памяти может не хватить для полноценной работы другой программе.

Несмотря на то что компьютеры сейчас оперируют гигабайтами оперативной памяти (в моем собственном ноутбуке ее 16 Гбайт, а в служебном — целых 32), мы должны обходиться с этим ресурсом максимально аккуратно. Излишнее расходование ресурсов компьютера не даст вам плюса как программисту и вашей программе как инструменту. Не знаю, как вы, а я запросто могу отказаться от использования программы, если она станет работать медленно и будет забирать слишком много памяти.

А что если наша стоянка решит расшириться в два раза и занять пространство соседней территории, т. е. сможет вмещать 100 машин? Наш массив окажется переполнен, 50 дополнительных машин в него просто не поместятся, и программа может завершиться ошибкой при добавлении 51-й машины.

Проблема решается с помощью *динамических массивов*. В .NET существует несколько классов, которые позволяют работать с данными динамического (изменяемого) размера: `Dictionary`, `List`, `Queue`, `SortedList`, `Stack`. Все эти классы объединили одним термином — *коллекции*.

Ранее уже отмечалось, что в программировании массивы, размер которых может изменяться, называют *списками* (List), так что это слово в названии классов сразу же может сказать вам о том, что перед нами класс, который не фиксирован в размере, и в него можно добавлять или удалять элементы.

Сейчас же мы познакомимся с классом `ArrayList`. Этот класс позволяет динамически добавлять и удалять память для хранения очередного элемента массива (списка) и использует столько памяти, сколько нужно, с небольшими затратами дополнительных ресурсов на поддержку динамики. Но эти затраты настолько невелики, что о них можно забыть.

Этот класс и другие коллекции .NET объявлены в пространстве имен `System.Collections`, поэтому не забудьте подключить это пространство с помощью оператора `using` к модулю, где будете использовать коллекции, поскольку это пространство понадобится для большинства примеров в этой главе:

```
using System.Collections;
```

Класс `ArrayList` хранит массив объектов класса `Object`. Так как этот класс является предком или одним из предков для всех классов, то это значит, что мы можем поместить в список объект абсолютно любого класса. Я даже скажу больше — вы можете помещать в такой массив элементы разных классов. Первый элемент может быть класса `Person`, второй элемент — класса `Object`, а третий вообще может оказаться формой. Такие массивы можно назвать *нетипизированными*, потому что они не привязаны жестко к определенному классу элементов.

Хранение разных объектов в массиве весьма опасно, и при получении объектов из него нужно быть очень осторожным, чтобы правильно интерпретировать класс, которому принадлежит объект.

Итак, допустим, что нам надо наделить наш класс `Person` возможностью хранения списка детей. Использовать статический массив тут невозможно, потому что неизвестно, сколько элементов нужно выделить. Если выделить только 3 элемента, то большое количество многодетных семей не смогут уместить своих детей в этом списке. Если выделить целых 5 или даже 10 элементов, то для большинства семей этот список будет заполнен процентов на 10 или 20. Бессмысленное расходование памяти нецелесообразно.

Такую проблему можно решить с помощью динамического массива. Возможный вариант решения показан в листинге 7.3.

Листинг 7.3. Реализация динамического массива для списка детей

```
ArrayList children = new ArrayList();

public int NumberOfChildren { get { return children.Count; } }

public void AddChild(string firstName, string lastName)
{
    children.Add(new Person(firstName, lastName));
}
```

```
public void DeleteChild(int index)
{
    if (index >= 0 && index < children.Count)
        children.RemoveAt(index);
}

public Person GetChild(int index)
{
    if (index >= 0 && index < children.Count)
        return (Person)children[index];
    return null;
}

public Person this[int index]
{
    get { return (Person)children[index]; }
}
```

В этом примере объявлена переменная класса `ArrayList`. Переменная инициализируется как любой другой класс, и мы нигде не указываем размер будущего массива. Он будет расти и уменьшаться автоматически, по мере добавления или удаления элементов.

Чтобы не делать переменную `Children` открытой, но при этом предоставить возможность внешним классам работать со списком детей, я написал три метода: `AddChild()`, `DeleteChild()` и `GetChild()` — для добавления ребенка (элемента списка) в список, удаления его и получения.

Самое интересное происходит в методе получения элемента списка `GetChild()`. Метод получает в качестве параметра индекс элемента, который нужно вернуть. Сначала я проверяю значение `index`, чтобы убедиться, что это значение больше нуля и меньше количества элементов. Если у нас в списке только один элемент, а пользователь попытается удалить десятый, то это может привести к ошибке. Для доступа к элементу в списке `ArrayList` следует написать индекс нужного элемента списка в квадратных скобках после имени переменной:

```
Children[index]
```

Точно так же мы получали элементы массива статичного размера. Поскольку элементы массива нетипизированы и приравниваются к классу `Object`, то для получения класса `Person` (а элементы именно этого класса хранятся у нас в списке) мы используем приведение типов, указывая нужный тип в скобках перед переменной.

Давайте посмотрим на самые интересные методы и свойства класса `ArrayList`:

- ☐ `Count` — свойство, которое позволяет узнать количество элементов в массиве;
- ☐ `Add()` — добавление в список элемента, переданного в качестве параметра;
- ☐ `AddRange()` — добавление в список содержимого массива, переданного в качестве параметра;

- ❑ `Remove()` — удаление элемента, объект которого указан в качестве параметра. Если такой объект не найден в списке, то удаления не произойдет;
- ❑ `RemoveAt()` — удаление элемента с индексом, переданным в качестве параметра;
- ❑ `Clear()` — очистка содержимого списка;
- ❑ `Contains()` — позволяет узнать, есть ли в списке элемент в виде объекта, указанного в качестве параметра;
- ❑ `Insert()` — вставляет элемент в указанную позицию.

Этот класс также наследует все свойства и методы базового для списков класса `Array`. Соответственно в нем есть методы сортировки, определения индекса элемента и т. д., потому что класс `Array` реализует такие интерфейсы, как `ICollection` и `IEnumerable`, о которых мы поговорим в *разд. 7.5*.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter7\ArrayListProject` сопровождающего книгу электронного архива (см. *приложение*).

7.4. Индексаторы массива

Давайте посмотрим, как сделать так, чтобы программист мог запрашивать всех детей класса `Person`, обращаясь непосредственно к нему. Это вполне возможная операция, но для начала нужно в этот класс добавить следующий код:

```
public int GetChildrenNumber()
{
    return Children.Count;
}

public Person this[int index]
{
    get { return (Person)Children[index]; }
}
```

Сначала мы добавили классу открытый метод `GetChildrenNumber()`, который возвращает количество элементов в списке. После этого добавляется интересное свойство с именем `this`. Мы уже знаем, что в C# такое ключевое слово указывает на текущий объект. Неужели так же можно называть еще и свойства? Можно, но не простые свойства, а *индексаторы*. Они получают параметры не в круглых скобках, а в квадратных. В нашем случае индексатор возвращает элемент массива, соответствующий элементу, запрошенному через переменную `index` из массива `Children`.

При обращении по индексу нужно быть осторожным, потому что если массив состоит из 10 объектов, а вы попытаетесь обратиться к элементу под индексом, превышающим 10, то приложение выдаст исключительную ситуацию `System.IndexOutOfRangeException`. Мы об этом еще не говорили, но в реальности это повлечет открытие окна, содержащего сообщение, что приложение выполнило недопус-

тимую операцию, и предложение пользователю закрыть приложение или продолжить работу. Если он работу продолжит, то дальнейшее поведение программы может быть любым.

Произвольный доступ к памяти за пределами массива является серьезной уязвимостью в программах на многих языках, но платформа .NET защищает нас и не дает выполнять такие обращения. Если же вы попытаетесь это сделать, то будет выдано исключение. Поэтому перед обращением к элементу массива лучше проверить, является ли индекс положительным числом и меньше ли он, чем количество элементов в массиве.

Теперь посмотрим, как можно использовать этот индексатор. Если у вас есть переменная `person` класса `Person`, то к объектам детей можно обратиться следующим образом:

```
person[индекс]
```

Несмотря на то что класс `Person` не является сам по себе массивом, он стал работать как массив, благодаря индексатору `this[int index]`. Индексатор просто возвращает содержимое массива `ArrayList`. Если переменную `children` у класса `Person` сделать открытой (`public`), то благодаря написанному нами индексатору следующие две строки будут идентичны:

```
person[индекс]  
person.children[индекс]
```

Допустим, вы хотите вывести на экран всех детей — для этого можно использовать следующий цикл:

```
for (int i = 0; i < person.NumberOfChildren; i++)  
{  
    Console.WriteLine(" " + person[i].FirstName + " " +  
        person[i].LastName);  
}
```

Не могу сказать, что этот код идеален. Переменная `person` — это объект, где хранятся данные человека, но когда мы обращаемся к `person[i]`, то обращаемся к детям этого человека. К сожалению, с точки зрения логики пример не самый удачный, но мне нужно было как-то показать индексаторы, вот я здесь их так и показал, хотя в реальных проектах к такому способу прибегать бы не стал.

7.5. Интерфейсы массивов

Существует несколько интерфейсов, которые позволяют получить при использовании массивов всю их мощь. Укажу основные из них:

- ❑ `ICollection` — коллекция определяет методы добавления элементов в массив, получения интерфейса перечисления элементов и определения количества элементов. Именно этот интерфейс служит для доступа к элементам массива таких компонентов, как `ListView`, `ListBox` и т. д.;

- ❑ `IComparer` — интерфейс, использующийся для сравнения элементов во время сортировки;
- ❑ `IDictionary` — интерфейс, позволяющий реализовать доступ к элементам по ключу/значению;
- ❑ `IEnumerable` — если класс реализует этот интерфейс, то объект этого класса можно использовать в операторе цикла `foreach`, т. е. он содержит необходимые методы, через которые можно перебирать элементы списка;
- ❑ `IDictionaryEnumerator` — содержит объявления методов, которые позволяют сделать интерфейс `IDictionary` перечисляемым. Этот интерфейс является наследником `IEnumerator`;
- ❑ `IList` — если класс реализует этот интерфейс, то к элементам его массива можно обращаться по индексу.

Далее я приведу описание ряда интересных решений по использованию интерфейсов в реальных приложениях. Интерфейсы при этом мы будем рассматривать не в том порядке, как они указаны в приведенном списке, а так, чтобы рассказ получился максимально связным.

7.5.1. Интерфейс *IEnumerator*

Если класс наследует интерфейс `IEnumerator`, то его можно использовать в цикле `foreach`. У нас есть класс `Person`, который благодаря индексатору может работать почти как массив. А что если мы попробуем перечислить всех детей с помощью цикла `foreach`? Этот код завершится ошибкой:

```
foreach (Person children in person)
    Console.WriteLine(children.FirstName);
```

Так как класс `Person` не реализует интерфейса `IEnumerator`, операция перечисления будет невозможной. Неужели ради цикла нам придется открывать свойство `children` и использовать его?

```
foreach (Person children in person.children)
    Console.WriteLine (children.FirstName);
```

Можно и так, потому что свойство `children` является объектом класса `ArrayList`, а циклы используют интерфейс `IEnumerator` для перечисления элементов. Этот код вполне корректен, но мы можем сделать так, чтобы и предыдущий код тоже работал. Для этого класс `Person` должен реализовать интерфейс `IEnumerator`. Этот интерфейс описывает всего один метод `GetEnumerator()`, который возвращает интерфейс `IEnumerator`. В нашем случае реализация этого метода может выглядеть следующим образом:

```
public IEnumerator GetEnumerator()
{
    return Children.GetEnumerator();
}
```

Поскольку индексатор перенаправляет обращение по индексу к массиву `Children`, то метод `GetEnumerator()` может поступить так же. Он просто может вызвать такой же метод списка `Children`, и следующий цикл `foreach` заработает:

```
foreach (Person children in person)
    Console.WriteLine (children.FirstName);
```

Да, теперь этот код отработает корректно, и программа будет скомпилирована. Но на самом деле мы всего лишь использовали готовую функцию другого класса, и класс `Person` становится здесь только лишь посредником. А как самому реализовать `IEnumerator`? Чтобы понять это, нужно посмотреть на возвращаемое значение метода `GetEnumerator()`. А возвращается в качестве значения `IEnumerator`. Методом сложнейшей дедукции можно догадаться, что символ `I` в начале названия говорит о том, что перед нами интерфейс. Значит, нам достаточно только реализовать его и вернуть свой вариант реализации.

Давайте создадим новый файл класса и назовем его `PersonEnumerator`. Этот класс будет реализовывать интерфейс перечисления (мой вариант можно увидеть в листинге 7.4).

Листинг 7.4. Класс `PersonEnumerator`

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;

namespace ArrayListProject
{
    public class PersonEnumerator: IEnumerator
    {
        int currIndex = -1;
        PersonClass.Person person;

        public PersonEnumerator(PersonClass.Person person)
        {
            this.person = person;
        }

        #region IEnumerator Members

        public object Current
        {
            get { return person[currIndex]; }
        }

        public bool MoveNext()
        {
            currIndex++;
        }
    }
}
```

```
        if (currIndex >= person.GetChildrenNumber())
            return false;
        else
            return true;
    }

    public void Reset()
    {
        currIndex = -1;
    }
    #endregion
}
```

Сразу же посмотрим, как можно использовать наш собственный класс. Для этого метод `GetEnumerator()` в классе `Person` нужно изменить следующим образом:

```
public IEnumerator GetEnumerator()
{
    return new PersonEnumerator(this);
}
```

Теперь метод возвращает результат создания экземпляра класса `PersonEnumerator`, который реализует интерфейс `IEnumerator`. Интерфейс `IEnumerator` описывает всего три метода, которые необходимы для создания перечисления:

- ❑ `Current()` — метод должен возвращать текущий элемент списка. Это значит, что мы где-то должны держать счетчик, который будет указывать, какой элемент сейчас является текущим;
- ❑ `MoveNext()` — изменить счетчик или ссылку на следующий элемент списка;
- ❑ `Reset()` — сбросить счетчик.

В нашей реализации класс `PersonEnumerator` получает в конструкторе экземпляр класса `Person`, и в методах, которые реализуют интерфейс `IEnumerator`, происходит перебор всех детей списка.

Вы можете написать реализацию интерфейса так, чтобы перебор детей выполнялся в обратном порядке. Для этого просто нужно начальное значение счетчика устанавливать в максимальный индекс списка и при сбросе тоже переставлять счетчик на конец списка. При этом метод `MoveNext()` должен не увеличивать индекс, а уменьшать его.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter7\IEnumerableProject` сопровождающего книгу электронного архива (см. приложение).

7.5.2. Интерфейсы *IComparer* и *IComparable*

Эти два интерфейса можно рассматривать совместно, потому что они предназначены для сортировки элементов массива. Когда массив содержит простые типы данных — например, числа, то .NET и без вас знает, как их сравнивать. Если же в списке находятся элементы пользовательского типа (классы), то логику их сравнения вы должны написать сами. Тут платформа просто не сможет догадаться, по какому принципу вы хотите реализовать сортировку и какие свойства классов нужно сравнивать.

Попробуйте сейчас написать в классе `Person` следующий метод:

```
public void SortChildren()
{
    Children.Sort();
}
```

Этот метод класса `Person` предназначен для сортировки объектов (детей). Попробуйте написать в тестовом приложении вызов этого метода:

```
Person person = new Person("Сергей", "Иванов");

person.AddChild("Сергей", "Иванов");
person.AddChild("Алексей", "Иванов");
person.AddChild("Валя", "Иванов");

person.SortChildren();
```

Попытка отсортировать массив завершится неудачей. Если вы запустите пример из среды разработки, то управление будет передано ей и вы увидите сообщение об исключительной ситуации (рис. 7.2). Ошибка гласит, что произошел сбой при сравнении двух элементов массива. Платформа просто не поняла, как производить сравнение: сначала по имени, а потом по фамилии или, может, по возрасту.

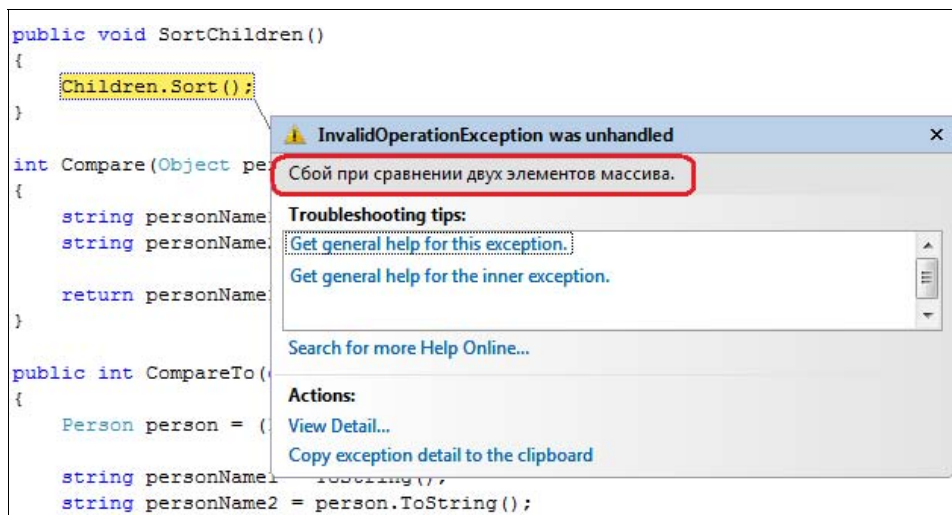


Рис. 7.2. Ошибка сравнения двух элементов пользовательского типа

В массиве находятся элементы класса `Person`. Чтобы объекты какого-либо класса могли участвовать в сортировке в массивах, такой класс должен реализовывать интерфейс `IComparable`. Этот интерфейс определяет только один метод — `CompareTo()`, который должен сравнивать текущий экземпляр класса с объектом, переданным в качестве параметра. В качестве результата метод должен вернуть число:

- ❑ если текущий объект меньше переданного объекта, то результат меньше нуля;
- ❑ если результат равен нулю, то объекты одинаковы;
- ❑ если результат положительный, то текущий объект больше переданного.

Добавьте сейчас интерфейс `IComparable` к нашему классу `Person`. Возможный метод сравнения `CompareTo()` может выглядеть так:

```
public int CompareTo(Object obj)
{
    Person person = (Person)obj;

    string personName1 = ToString();
    string personName2 = person.ToString();

    return personName1.CompareTo(personName2.ToString());
}
```

В качестве параметра методу будет передаваться объект класса `Person`, но в описании интерфейса нам дается класс `Object`. Чтобы было удобнее, в методе заведена переменная класса `Person`, куда сохраняется приведенная переменная `obj`.

Теперь для еще одного удобства создаются две переменные. В первую переменную сохраняется текущий объект, приведенный к строке с помощью метода `ToString()`, а во вторую — сохраняется переданный объект, приведенный к строке таким же методом. Я тут подразумеваю, что будет использоваться переопределенный нами в *разд. 3.9* метод `ToString()`, который выглядит так:

```
public new string ToString()
{
    return FirstName + " " + LastName;
}
```

Поскольку метод `ToString` объявлен как виртуальный, можно использовать `override` вместо `new`, но я решил в нашем случае применить именно `new`.

Получается, что в переменных `personName1` и `personName2` будут находиться строки, содержащие имя и фамилию, сложенные через пробел. Именно эти строки мы и сравниваем в последней строке метода. Для этого используется метод `CompareTo()`, который есть у строки. Обратите внимание на название метода! Да, это реализация метода для интерфейса `IComparable`. Получается, что у строки есть метод интерфейса, и поэтому массивы строк сортируются без проблем. Мы же используем его в своем методе сравнения.

Сравнивать по имени и фамилии в одной строке — не самый удачный выбор: лучше сравнивать по каждому из полей отдельно. Давайте отсортируем сначала по фамилии, а потом по имени:

```
public int CompareTo(Object obj)
{
    Person person = (Person)obj;

    int result = this.LastName.CompareTo(person.LastName);
    if (result == 0) {
        return this.FirstName.CompareTo(person.FirstName);
    }
    return result;
}
```

Здесь я уменьшил количество промежуточных переменных и сразу использую объекты. Сначала происходит сравнение по фамилии. Если результат равен 0 (фамилии одинаковы), то возвращаем результат сравнения по имени. Иначе результатом будет сравнение фамилий из переменной `result`.

Теперь мы реализовали нужный метод, и если сейчас запустить метод сортировки массива детей, то он отработает без проблем.

А что если у класса есть реализация метода сравнения, но мы хотим выполнить свою сортировку на основе собственного алгоритма? Ну, например, собираемся сравнивать только имена или только фамилии. А может, мы добавим в класс поле возраста и будем сравнивать еще и его. Как поступить в этом случае? Можно создать наследника от `Person` и переопределить метод сравнения, но это не очень хорошее решение. Только ради того, чтобы создать новый алгоритм сравнения, создавать наследника неэффективно. Есть выход лучше — интерфейс `IComparer`.

У метода `Sort()` массивов есть перегруженный вариант, который получает в качестве параметра интерфейс `IComparer`. Этот интерфейс определяет один метод `Compare()`, получающий два объекта, которые нужно сравнить. В качестве результата возвращается число, как и у `CompareTo()`. Возможный вариант для нашего класса может выглядеть следующим образом:

```
int IComparer.Compare(Object person1, Object person2)
{
    string personName1 = ((Person)person1).ToString();
    string personName2 = ((Person)person2).ToString();

    return personName1.CompareTo(personName2);
}
```

Чтобы использовать этот метод, нужно вызвать метод `Sort()`, указав ему в качестве параметра объект, который реализует интерфейс `IComparer`. Поскольку мы реализовали этот интерфейс прямо в классе `Person`, метод можно вызвать так:

```
Children.Sort(this);
```

Но это не обязательно — в качестве параметра может быть передан любой другой класс, лишь бы он реализовывал `IComparer` и умел сравнивать объекты нашего класса. Посмотрим это на примере. Давайте создадим класс, который будет сортировать массив в обратном порядке. Да, это слишком простой пример, но наглядный:

```
class SortTest : IComparer
{
    int IComparer.Compare(Object person1, Object person2)
    {
        string personName1 = ((Person)person1).ToString();
        string personName2 = ((Person)person2).ToString();
        return personName2.CompareTo(personName1);
    }
}
```

Класс `SortTest` реализует интерфейс `IComparer`, и его метод похож на тот, что мы уже рассматривали. Разница заключается в строке сравнения. В нашем случае `CompareTo()` вызывается для второго человека, а не для первого, поэтому результат будет отсортирован в обратном порядке. Теперь этот класс можно использовать для сортировки, например, так:

```
Children.Sort(new SortTest());
```

Методу `Sort` передается экземпляр класса `SortTest`, который и будет сортировать данные массива `Children`. В этом примере создается экземпляр класса `SortTest` и сразу же — без сохранения в отдельной переменной — передается методу, потому что мы не будем больше использовать этот объект, так что и заводить переменную я не вижу смысла.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter7\SortProject` сопровождающего книгу электронного архива (см. приложение).

7.6. Оператор *yield*

В C# есть еще один способ реализовать метод `GetEnumerator()` и при этом даже не реализовывать `IEnumerator`. Для этого используется оператор `yield`. С его участием метод `GetEnumerator()` будет выглядеть следующим образом:

```
public IEnumerator GetEnumerator()
{
    foreach (Person p in Children)
    {
        yield return c;
    }
}
```

В таком варианте метода `GetEnumerator()` мы запускаем цикл, который перебирает все элементы списка. В нашем случае так происходит в еще одном цикле `foreach`, но, в зависимости от приложения, это может быть реализовано по вашему усмотрению. Внутри цикла выполняется конструкция `yield return`. Это не просто `return`, который прерывает выполнение метода и возвращает значение, это `yield return`,

который возвращает указанное значение внешнему итератору (циклу `foreach`) и продолжает работать без прерывания работы метода.

Тут нужно заметить, что `yield return` не может использоваться внутри блоков обработки исключительных ситуаций, о которых мы будем говорить в главе 9.

7.7. Стандартные списки

В .NET существует несколько классов для работы с динамическими массивами, и все они описаны в пространстве имен `System.Collections`. С одним из этих классов мы уже познакомились — это `ArrayList`, который предоставляет нам возможность создавать массив с динамически изменяемым размером. Кроме того, в этом же пространстве имен можно найти следующие классы:

- ❑ `BitArray` — позволяет создать компактный массив битовых значений. Каждый элемент списка может принимать значения только `true` или `false`;
- ❑ `Hashtable` — коллекция, в которой данные хранятся в виде пары ключ/значение. Доступ к значениям в коллекции происходит по ключу и достаточно быстро;
- ❑ `Queue` — этот класс реализует список по принципу FIFO (`first-in, first-out` — первым пришел, первым ушел). Это значит, что элементы добавляются в конец списка, а снимаются только из начала;
- ❑ `SortedList` — список, в котором данные хранятся в отсортированном виде. Такие списки удобны, когда нужно реализовывать бинарный поиск или просто надо работать с данными как в словаре;
- ❑ `Stack` — эти списки строятся по принципу LIFO (`last-in, first-out` — последним пришел, первым ушел).

Наиболее интересными для нас сейчас могут быть классы `Queue`, `Stack` и `Hashtable`, потому что работа с ними немного отличается от остальных. Списки `SortedList` и `BitArray` ближе к `ArrayList`, с которым мы уже знакомы, и тут проблем возникать не должно.

7.7.1. Класс *Queue*

Этот тип коллекции позволяет реализовать список в виде очереди, где элементы обрабатываются в порядке поступления. Мы сталкиваемся с такой обработкой каждый день в нашей реальной жизни, и коллекция очереди идентична по идеологии с очередью в магазине. Ведь это логично и честно, что чем раньше пришел человек, тем раньше его должны обслужить.

У класса `Queue` нет привычных для коллекций методов типа `Add()` и `Remove()`, потому что эти методы должны иметь возможность доступа к любому элементу списка, а это нарушает принцип очереди. Чтобы не нарушать этот принцип, у класса есть следующие три метода для работы с элементами:

- ❑ `Enqueue()` — добавить указанный в качестве параметра элемент в очередь. Элемент будет добавлен в конец списка;

- ❑ `Dequeue()` — вернуть очередной элемент очереди из списка и одновременно удалить его. Будет возвращен первый элемент списка;
- ❑ `Peek()` — вернуть очередной элемент очереди без удаления.

Следующий код показывает небольшой пример использования очереди:

```
Queue queue = new Queue();
queue.Enqueue("Первый");
queue.Enqueue("Второй");
queue.Enqueue("Третий");
queue.Enqueue("Четвертый");
queue.Enqueue("Пятый");

do
{
    String s = queue.Dequeue().ToString();
    Console.WriteLine(s);
} while (queue.Count > 0);
```

В этом примере создается экземпляр класса `Queue`, который наполняется пятью элементами. После этого запускается цикл, который выводит на экран все элементы. Цикл выполняется, пока в списке есть элементы.

Тут нужно быть внимательным при построении циклов, потому что метод `Dequeue()` сокращает размер списка на единицу. Это значит, что цикл `foreach` может завершиться ошибкой, да и цикл `for` тоже может выйти за пределы списка. Цикл `for` может выглядеть следующим образом:

```
for ( ; queue.Count > 0; )
{
    String s = queue.Dequeue().ToString();
    Console.WriteLine(s);
}
```

Первый и последний параметры цикла пустые. Достаточно только второго параметра, который проверяет количество значений в списке.

7.7.2. Класс *Stack*

Как и очередь, этот список не должен предоставлять возможности доступа к произвольному элементу, поэтому у него также есть свои методы для того, чтобы поместить объект в стек и снять очередной объект:

- ❑ `Push()` — поместить указанный в качестве параметра объект в стек;
- ❑ `Pop()` — снять последний добавленный в стек объект, т. е. вернуть объект с одновременным удалением;
- ❑ `Peek()` — вернуть очередной элемент без удаления его из списка.

Следующий код показывает пример использования стека:

```
Stack stack = new Stack();
stack.Push("Первый");
stack.Push("Второй");
stack.Push("Третий");
stack.Push("Четвертый");
stack.Push("Пятый");

for (; stack.Count > 0; )
{
    String s = stack.Pop().ToString();
    Console.WriteLine(s);
}
```

В отличие от примера с `Dequeue()`, в этом случае элементы будут выведены в обратном порядке их помещения в список. В остальном классы очень похожи по методу работы с ними.

7.7.3. Класс *Hashtable*

На самом деле этот класс хранит целых два списка: список ключей и список значений. Первый из них хранится в свойстве `Keys`, а второй — в свойстве `Values`. Эти два списка вы можете просматривать независимо, а можете использовать значения из списка ключей для доступа к значениям из списка значений.

Для добавления значений в оба списка используется всего один метод: `Add()`. Он принимает в качестве значений два параметра: ключ и соответствующее ему значение. И то и другое имеет тип `Object`, значит, они могут быть абсолютно любого типа данных.

Хеш-таблица — это массив, в котором хранятся *хеши*. Я понимаю, что это очень логичное объяснение и абсолютно бесполезное, поэтому попробую изъясниться чуть подробнее. Так что такое хеш? Это некое преобразование данных. Самый простой способ получить хеш — использовать деление на какое-нибудь число и взять при этом только целую часть. Так мы можем уплотнить данные в очень маленький массив.

Допустим, что у нас есть массив, данные которого могут изменяться от 0 до 10 000. При этом мы знаем, что массив состоит из 100 элементов. Как сделать так, чтобы получить максимально быстрый способ доступа к любому элементу? Самый быстрый способ — создать массив из 10 000 элементов и помещать в него данные в соответствии со значением. Например, число 5 нужно поместить в пятую позицию массива. Теперь, чтобы найти любое число, нужно просто обратиться к элементу массива по его номеру (индексу). Если значение по индексу нулевое, то, значит, в массиве просто нет такого значения.

Но у нас вариантов значений 10 000, а заполненных элементов всего 100. Не кажется ли вам, что это слишком расточительный расход памяти? Я считаю, что так оно

и есть. Проблему решает хеш. Прежде чем помещать значение в таблицу, просто делим его на количество элементов в массиве (100) и получаем позицию элемента в хеш-таблице. Так мы создаем массив только из 100 элементов и уплотняем в него значения.

Но ведь и $1/100$ будет представлена нулем, и $2/1000$ тоже — не забываем, что мы берем только целую часть. Получается, что оба значения придется поместить в нулевую позицию. Как решить эту проблему? Каждый элемент хеш-таблицы можно представить в виде списка (`List`), и тогда оба значения без проблем поместятся в нулевом элементе.

Это лишь пример того, как может быть реализован простой метод хеш-таблицы. Я не смотрел на внутренности реализации Microsoft, но, судя по скорости работы, они используют весьма эффективный метод.

Возможный вариант работы с хеш-таблицами можно увидеть в листинге 7.5.

Листинг 7.5. Пример работы с хеш-таблицами

```
Hashtable hash = new Hashtable();
hash.Add("Михаил Смирнов", new Person("Михаил", "Смирнов"));
hash.Add("Сергей Иванов", new Person("Сергей", "Иванов"));
hash.Add("Алексей Петров", new Person("Алексей", "Петров"));

Console.WriteLine("Значения:");
foreach (Person p in hash.Values)
    Console.WriteLine(p.LastName);
Console.WriteLine("\nКлючи:");
foreach (String s in hash.Keys)
    Console.WriteLine(s);

Console.WriteLine("\nДоступ к значению по ключу:");
foreach (Object key in hash.Keys)
{
    Person p = (Person)hash[key];
    Console.WriteLine("Ключ: '" + key + "' Значение:" + p.FirstName);
}
```

Здесь сначала создается новый объект класса `Hashtable` и в него добавляются три пары ключ/значение. В качестве ключа выступает строка, а в качестве значения — объект класса `Person`. Теперь, чтобы получить объект по его ключу, можно использовать следующую строку:

```
hash["Михаил Смирнов"]
```

Обращение происходит точно так же, как и с простыми массивами, когда в квадратных скобках мы указывали индекс нужного нам элемента. В случае с `Hashtable` мы указываем не индекс, а ключ, который был задан при создании элемента.

После заполнения списка я показал вам, как можно перечислить:

- ☐ все значения в списке;
- ☐ все ключи в списке;
- ☐ все ключи в списке и соответствующие им значения.

Хеш-таблицы являются основой оптимизации для достаточно большого количества алгоритмов, поскольку они очень удобны, когда нужно хранить где-то два связанных массива, и они также весьма производительны с точки зрения доступа к элементам. Чтобы получить какое-то значение, мы просто обращаемся к таблице по ключу и получаем его достаточно быстро. Если использовать простой список (массив), то для получения нужного элемента придется перебирать в его поиске все значения.

Работая с хеш-таблицами, нужно быть осторожным. Если попытаться добавить в таблицу ключ, который уже существует в ней, то произойдет ошибка. Например, здесь ошибка во второй строке, потому что я пытаюсь добавить один и тот же ключ дважды:

```
hash.Add("Алексей Петров", new Person("Алексей", "Петров"));  
hash.Add("Алексей Петров", new Person("Алексей", "Петров"));
```

Если есть шанс, что в таблице уже имеется ключ, то нужно сначала проверить наличие ключа с помощью метода `ContainsKey`. Метод возвращает истину, если в таблице уже есть ключ, имя которого передано в качестве параметра. Чтобы перевернуть проверку, можно просто поставить символ `!` перед методом.

В следующем примере значение будет добавлено, только если ключа нет в таблице:

```
if (!hash.ContainsKey("Алексей Петров"))  
    hash.Add("Алексей Петров", new Person("Алексей", "Петров"));
```

Если надо изменить значение, это делается так же, как и в случае с массивами:

```
hash["Алексей Петров"] = new Person("Алексей", "Петров");
```

То есть можно сделать логику: если ключа нет в таблице, то добавить, иначе изменить:

```
if (!hash.ContainsKey("Алексей Петров"))  
    hash.Add("Алексей Петров", new Person("Алексей", "Петров"));  
else  
    hash["Алексей Петров"] = new Person("Алексей", "Петров");
```

7.8. Типизированные массивы

Неудобство всех массивов из пространства имен `System.Collections`, которые мы рассмотрели в *разд. 7.7*, заключается в том, что они хранят нетипизированные данные. Любой элемент массива представляет собой экземпляр класса `Object`. Так как это базовый тип, то мы можем привести к нему абсолютно любые объекты и хранить в массиве все, что угодно. Но это «что угодно» не является безопасным спосо-

бом программирования. Если в массив поместить разные объекты, то приведение может оказаться проблематичным и вести к ошибкам в программах. Поэтому приходится быть аккуратным.

Когда в массиве должны использоваться объекты строго определенного типа, то лучше создать массив для хранения элементов только этого типа, чтобы вы не могли поместить в список что-либо иное. Так доступ к элементам может быть упрощен, а отсутствие необходимости приведения сокращает вероятность появления ошибок.

Существует несколько классов, с помощью которых можно создать списки различного типа для хранения объектов любого класса:

- ❑ `List` — наиболее популярный класс для работы с однонаправленным списком. Если вы не знаете, что выбрать, то лучше начать с использования этого класса;
- ❑ `LinkedList` — двунаправленный список, в котором каждый элемент списка имеет ссылку не только на следующий элемент в списке, но и на предыдущий;
- ❑ `Stack` — классический стек;
- ❑ `Queue` — классическая очередь.

Это основные классы, с которыми мне приходилось сталкиваться, хотя чаще я работаю именно с первым классом.

Для того чтобы создать строго типизированный список одного из указанных классов, используется следующий формат:

```
Список<тип> переменная;
```

Классы для работы с типизированными массивами находятся в пространстве имен `System.Collections.Generic`, так что не забудьте подключить его в начале файла:

```
using System.Collections.Generic;
```

В этом случае параметр *список* можно заменить на любой класс списков, которые мы рассмотрели, а параметр *тип* — на любой тип данных.

Такую переменную нужно инициализировать, и это делается следующим образом:

```
Список<тип> переменная = new Список<тип>();
```

Ничего не напоминает вам? Это шаблоны — когда мы объявляем класс:

```
class List<T> {  
  
}
```

где *T* — любой тип данных.

Все это очень похоже на использование любого другого класса списка, просто после названия класса в угловых скобках указывается конкретный тип данных, который будет храниться в списке.

Мы далее рассмотрим строго типизированные списки на основе класса `List`, а работа с остальными классами практически идентична.

Итак, допустим, что нам нужно хранить список объектов `Person`. Чтобы объявить динамический массив строгого типа для хранения объектов этого класса, нужно написать следующую строку:

```
List<Person> persons = new List<Person>();
```

Посмотрим на пример из листинга 7.6.

Листинг 7.6. Использование строго типизированного массива

```
List<Person> persons = new List<Person>();

// заполняем массив
persons.Add(new Person("Иван", "Иванов"));
persons.Add(new Person("Сергей", "Петров"));
persons.Add(new Person("Игорь", "Сидоров"));

// изменяем имя нулевого человека
persons[0].FirstName = "Новое имя";

// вывод содержимого списка
foreach (Person p in persons)
    Console.WriteLine(p.FirstName + " " + p.LastName);
Console.ReadLine();
```

В этом примере создается строго типизированный список `persons` для хранения объектов класса `Person`. Затем в список добавляются три экземпляра класса `Person`. Самое интересное начинается после этого — в цикле `foreach` перебираются все элементы массива и выводятся имена и фамилии людей. При этом не нужно приводить тип данных. Когда мы изменяем имя нулевого человека, то не происходит никакого приведения типа данных.

Если попробовать добавить в массив любой другой тип данных, отличный от `Person`, то компилятор выдаст ошибку и не даст откомпилировать код. Например, завершится ошибкой попытка добавить строку:

```
persons.Add("Строка");
```

Если надо обратиться к элементам списка, нам не потребуется приводить типы данных, потому что перед нами строго типизированный список, и ничего, кроме объектов класса `Person`, там не может быть.

Тут есть один нюанс — наследственность. В типизированный список можно добавлять не только определенный тип, но и любых его наследников. Например, в список `List<Person>` можно добавлять не только объекты класса `Person`, но и любые объекты классов, которые будут являться наследниками от `Person`.

Чтобы продемонстрировать это, попробуйте поменять объявление списка в листинге 7.6 на следующее:

```
List<Object> persons = new List<Object>();
```

Так как `Object` является базовым для всех, то в такой список можно добавить и объекты `Person`, которые есть в этом примере. Значит, с добавлением никаких проблем не возникнет, зато возникнут проблемы со следующей строкой:

```
persons[0].FirstName = "Новое имя";
```

Поскольку на этот раз список объявлен как хранилище для объектов класса `Object`, то понадобится приведение типов, несмотря на то, что реально в списке хранятся объекты `Person`:

```
((Person)persons[0]).FirstName = "Новое имя";
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter7\TypedArray` сопровождающего книгу электронного архива (см. приложение).

Старайтесь по возможности использовать типизированные версии массивов, потому что они защищают нас от возможных проблем добавления неверного объекта или ошибки при чтении.

7.9. Словарь

Мы уже знакомы с хешами, которые отлично подходят для хранения ключей/значений. Для работы с типизированными коллекциями в C# сделали немного другой тип — *словарь*. В принципе, хеши реально схожи по своей идее со словарями. Например, в англо-русском словаре ключом выступало бы английское слово, а перевод, транскрипцию и все остальное можно поместить в значение.

Словарь `Dictionary` отличается от `Hashtable` тем, что ему можно задать тип данных для ключа и тип для значения.

Вот так можно было бы создать типизированную версию хеша с помощью словаря:

```
Dictionary<string, Person> dict = new Dictionary<string, Person>();
```

На самом деле `Dictionary` работает как хеш, просто типизированный. В угловых скобках нужно указать два типа данных: первый тип — для ключа, второй тип — для значения. В качестве ключа, как и в прошлый раз, я решил использовать строку.

Вот пример использования словаря:

```
Dictionary<string, Person> dict = new Dictionary<string, Person>();  
dict.Add("Бася", new Person("Василий", "Иванович"));  
dict.Add("Петя", new Person("Петр", "Сергеевич"));  
foreach (string key in dict.Keys)  
{  
    Console.WriteLine(dict[key]);  
}  
Console.ReadLine();
```

Здесь все выглядит и работает как `Hashtable`. Значения ключей находятся в свойстве `Keys`, а значения — в свойстве `Values`.

Как и в случае с хешами, в словарь нельзя добавлять значения с одним и тем же ключом. В следующем примере произойдет ошибка на попытке второй раз добавить значение:

```
dict.Add("Петя", new Person("Петр", "Сергеевич"));
dict.Add("Петя", new Person("Петр", "Сергеевич"));
```

Существование ключа в таблице можно проверить с помощью метода `ContainsKey`. В следующем примере если ключ существует, то он обновляется, иначе добавляется:

```
if (dict.ContainsKey("Петя"))
    dict["Петя"] = new Person("Петр", "Сергеевич");
else
    dict.Add("Петя", new Person("Петр", "Сергеевич"));
```

Я в этот раз сделал более прямую логику — без использования символа `НЕ (!)`. Прямую логику читать обычно проще.

Для удаления значения из массива используется метод `Remove`, которому нужно передать имя ключа, который вы хотите удалить:

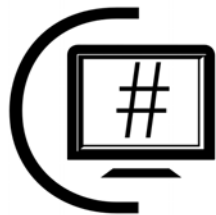
```
dict.Remove("Петя");
```

У словаря есть и другие методы. И так как я не ставил целью написать собственную версию MSDN, а показать принципы работы и научить именно работе с `C#` и классами фреймворка, то на этом пора закруглиться.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter7\Dict` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 8



Обработка исключительных ситуаций

Программисты очень часто пишут код, считая, что программа станет работать корректно, а окружение будет стабильно. Но, к сожалению, это далеко не всегда так.

Рассмотрим классическую задачу сохранения информации, когда программе нужно открыть файл, записать в него информацию и закрыть файл. Что тут сложного, когда в языке программирования есть все необходимое? А сложность возникает тогда, когда окружение оказывается нестабильным. Например, пользователь ввел несуществующий путь для файла или указал в имени файла недопустимые символы. В этом случае вызов команды создания/открытия файла завершится неудачей, и если эту неудачу пропустить сквозь пальцы, то программа может рухнуть.

Даже если имя файла вы проверили, и оно в порядке, и система смогла открыть его, запись в файл также может завершиться неудачей по множеству причин. Например, это оказался внешний носитель (USB-накопитель), который просто раньше, чем надо, выдернули из разъема, или свободного места там недостаточно, а значит, программа не сможет сохранить все необходимые данные в файл, и снова произойдет крушение, если не отработать эти внештатные ситуации и не отреагировать на них должным образом.

Когда происходит какая-то нештатная ситуация, приложение генерирует *исключение* (Exception), и если на него правильно не отреагировать, то приложение завершится аварийно. Реагировать на проблемные ситуации — наша задача. Если выдернуть USB-флешку, когда программа пытается сохранить на нее файл, как реагировать операционной системе? Продолжить выполнение? Но это может привести к серьезным проблемам, поскольку программа, не зная, что ей в такой ситуации делать, может зависнуть, да и весь компьютер «подвесить»... Чтобы программа не зависла, а продолжила работать, наша задача — перехватить ошибку и восстановить корректное состояние.

Все мы люди, и всем нам свойственно ошибаться, поэтому к проблемам работы программы может привести и несовершенство исходного кода или логики выполнения. Во всех этих случаях нам на помощь может прийти механизм обработки исключительных ситуаций.

8.1. Исключительные ситуации

Прежде чем мы приступим к рассмотрению механизма обработки исключительных ситуаций в .NET, я хочу еще немного времени потратить на лирику про ошибки и возникновение исключительных ситуаций, чтобы вы лучше понимали, когда нужно задействовать этот механизм. В случае с разного рода накопителями все понятно. Они не стабильны, в них могут появляться плохие блоки, может закончиться место, или устройство может быть отключено пользователем в самый неподходящий момент. Так что в процессе работы с такими устройствами мы должны отслеживать результат работы и реагировать на любые внештатные ситуации.

Но давайте посмотрим на следующий код:

```
double MyMul(int x, int y)
{
    return x / y;
}
```

Что в нем такого страшного? А страшным в этом коде является деление. В классической математике деление на ноль невозможно. Это в высшей математике нуля нет, а есть бесконечно малое число, при делении на которое появляется бесконечно большое число. Компьютер думает классически, а значит, при попытке разделить на ноль сгенерирует ошибку.

Случай с делением на ноль я бы отнес к ошибкам логики программы. Всегда, где есть вероятность деления на ноль, желательно сначала проверить значение переменной:

```
double MyMul(int x, int y)
{
    if (y == 0)
    {
        Console.WriteLine("Ошибочка");
        return 0;
    }
    return x / y;
}
```

Эту проверку сделать не сложно, и использовать что-либо, кроме проверки на ноль, я не вижу смысла. Это лично мое мнение. А вот при работе с накопителем или любыми другими ресурсами, которые зависят от окружения, лучше задействовать механизм обработки исключений.

А как мы должны обрабатывать данные, вводимые пользователем? Мой опыт написания программ говорит, что далеко не всегда можно ограничиться проверками, чтобы обезопасить работу программы. На любую нашу гениальную идею по проверке корректности ввода некоторые пользователи умудряются ответить такими непредсказуемыми действиями, что и не знаешь, как на это отреагировать.

Чтобы проще было понять природу исключительных ситуаций, нужно знать, какие могут встретиться ошибки. Я попробовал классифицировать их следующим образом:

1. *Ошибки при работе с ресурсами компьютера, необходимыми программе.*

Яркий пример такой проблемы мы уже рассмотрели — это доступность жесткого диска или внешнего носителя. Много лет назад существовала и другая проблема — объем оперативной памяти. Сейчас компьютеры имеют уже достаточно оперативной памяти, а ОС Windows умеет эффективно использовать файл подкачки, поэтому об этой проблеме многие забывают. Но если вы хотите выделить уж очень большой массив данных или просто блок памяти размером более гигабайта напрямую от ОС, то я бы задумался о возможных в этих случаях исключительных ситуациях. Ошибки ресурсов необходимо отлавливать исключительными ситуациями, и это очень удобно.

2. *Ошибки логики приложений.*

Бывают случаи, когда программист неправильно просчитал все варианты выполнения программы, и какой-то из вариантов привел к тому, что программа выполнила недопустимую операцию, — например, деление на ноль, выход за пределы массива, использование неинициализированной переменной и т. д. Такие ошибки достаточно опасны, но просто глушить их механизмом обработки исключений неправильно. Логические ошибки программы должны жестко отлавливаться системой, потому что попытки продолжить работу после подобной внештатной ситуации опасны. Исключительные ситуации могут быть только хорошими помощниками для вылавливания проблем с логикой, но не их решением.

3. *Пользовательские ошибки.*

Они могут быть результатом неправильных действий пользователя или некорректного ввода данных. Первый вариант с некорректными действиями ближе к логическим ошибкам. Если пользователь смог выполнить некорректную последовательность действий, то виновата логика программы, которая допустила такую последовательность. Эти ошибки нужно отлавливать и исправлять. А вот защищаться от некорректного ввода данных лучше двумя способами одновременно: проверкой вводимых данных на допустимость и обработкой исключительных ситуаций.

В ОС Windows нет жесткого механизма обработки исключительных ситуаций, и разные библиотеки и программы по-разному реагируют на проблемные ситуации и по-разному информируют о них пользователя. Например, программисты на C++ при вызове разных функций могут получать числовые константы, которые возвращают ошибки, а для получения подробной информации об ошибке может использоваться что угодно — например, функция `GetLastError()`. Впрочем, такая функция вообще может отсутствовать, и тогда остается только искать документацию у производителя.

В .NET весь бардак с генерацией ошибок превратился в исключительные ситуации, которые теперь очень легко и удобно обрабатывать. При этом вы получаете всю необходимую информацию об ошибке, чтобы можно было корректно отреагировать на проблему без краха всего приложения.

8.2. Исключения в C#

Исключения в C# строятся на основе четырех ключевых слов: `try`, `catch`, `throw` и `finally`, а также классов исключительных ситуаций. Все исключения в .NET так или иначе происходят от класса `Exception` из пространства имен `System` (`System.Exception`). Я думаю, что нам нужно познакомиться с классом `Exception` более подробно, прежде чем двигаться дальше, — ведь этот класс является базовым для всех классов исключений, а значит, его свойства и методы наследуются всеми.

Итак, в составе `Exception` я бы выделил самое интересное:

- ❑ `Data` — коллекция в виде интерфейса `IDictionary`, в которой в виде списка ключ/значение пользователю дается подробная информация об исключении;
- ❑ `HelpLink` — может возвращать интернет-адрес (URL) файла справки с дополнительной информацией по этому классу исключений;
- ❑ `InnerException` — информация о внутренних исключениях, которые стали причиной той или иной исключительной ситуации;
- ❑ `Message` — короткое, но в большинстве случаев понятное текстовое описание ошибки;
- ❑ `Source` — имя сборки, сгенерировавшей исключение;
- ❑ `StackTrace` — строка, содержащая последовательность вызовов, которые привели к ошибке. Это свойство может быть полезно с точки зрения отладки кода и исключения возникшей проблемы.

Давайте посмотрим на исключительные ситуации поближе. Для этого возьмем классическую задачу превращения строки в число. Мне достаточно часто приходится в окнах устанавливать поля ввода или читать данные с носителей, а потом превращать их в число. Создадим консольное приложение и напишем в методе `Main()` этого приложения следующий код:

```
while (true)
{
    Console.WriteLine("Введите число");

    string inLine = Console.ReadLine();
    if (inLine == "q")
        break;

    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
```

Здесь запускается цикл `while`, который должен выполняться, пока условие в скобках не станет равным `false`. Так как условия нет, а просто стоит `true`, то такой цикл может выполняться вечно (бесконечный цикл). Единственный способ его прервать — явно написать оператор `break` внутри цикла.

Внутри цикла мы предлагаем пользователю ввести число. Затем считываем строки и проверяем, что было введено. Если это буква `d`, то прерываем работу цикла, иначе пытаемся превратить строку в число. Запустите программу и попробуйте ввести числа. Все будет работать прекрасно, пока вы не введете символ или слишком большое число, которое не может быть преобразовано в тип данных `Int32`. Попробуйте ввести одну букву и нажать клавишу `<Enter>`. В результате вы увидите сообщение об ошибке (рис. 8.1), и программа завершит работу аварийно (если запустите программу не из среды разработки и не в режиме отладки).

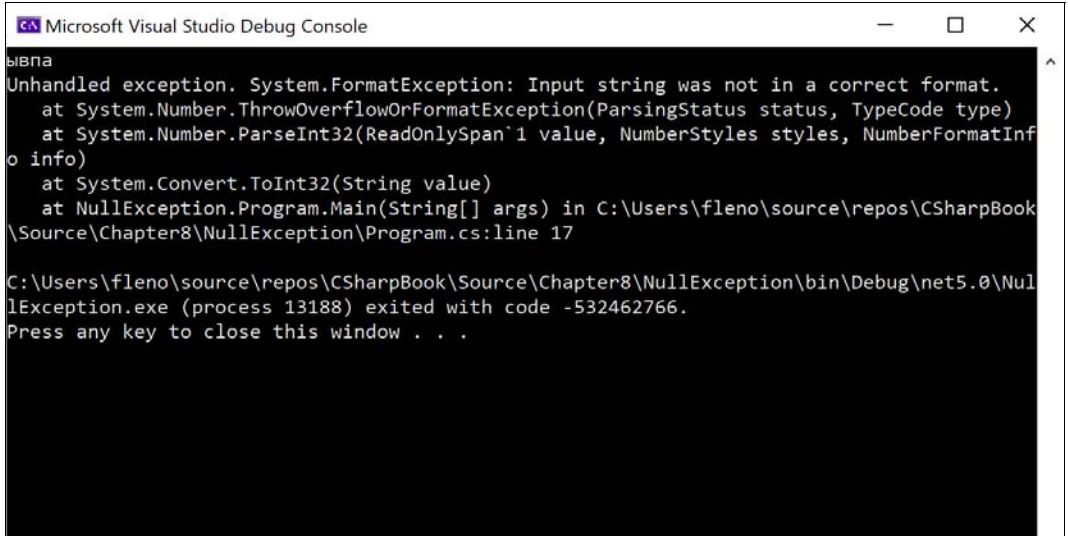


Рис. 8.1. Реакция на исключительную ситуацию при выполнении консольного приложения

Если запускать приложение из среды разработки, то в момент возникновения исключительной ситуации среда перехватит управление на себя и покажет нам место с ошибкой и класс ошибки (рис. 8.2). Нажав на ссылку **View Detail**, мы увидим дополнительное окно с подробной информацией об ошибке.

Как поступить в нашем случае? Понадеяться, что пользователь будет вводить только числа и не превысит максимального значения, или производить проверку строки? Я думаю, что большинство программистов используют тут исключительные ситуации, и это вполне приемлемый подход. Самый простой способ отловить проблемный код — заключить его в блок `try`:

```

try
{
    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
catch (Exception fe)
{
    Console.WriteLine(fe.Message);
}
  
```

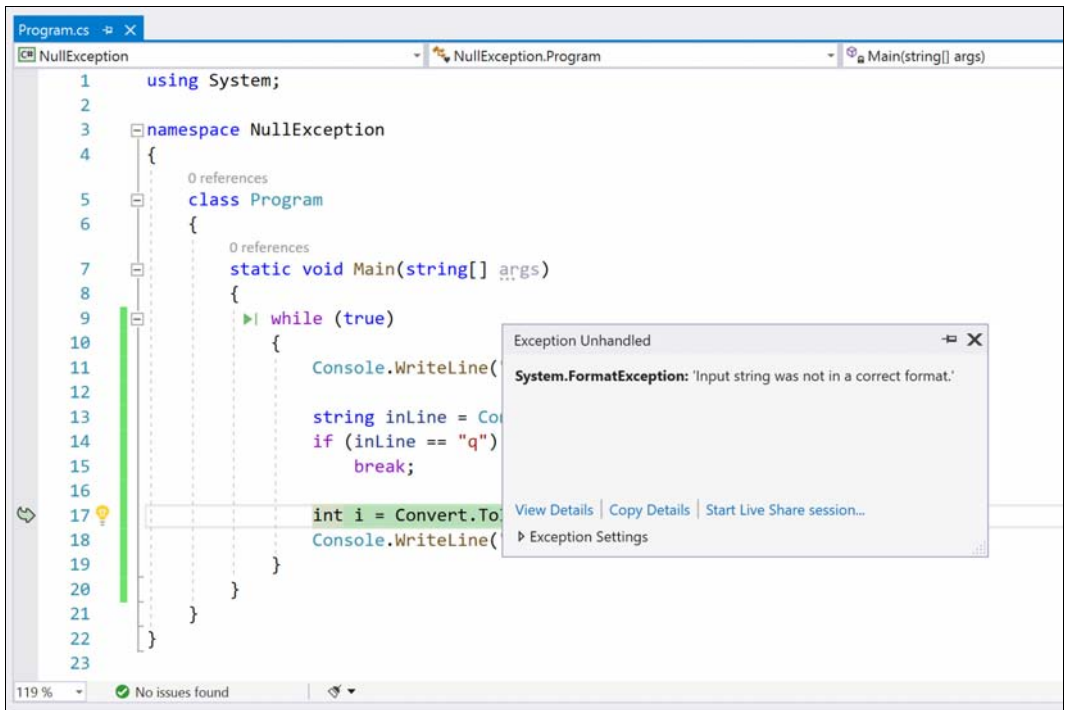


Рис. 8.2. Visual Studio перехватила управление на себя при исключительной ситуации

Строка, которая может привести к ошибке (в нашем случае к ошибке конвертирования), заключена в фигурные скобки `try`. Если внутри блока `try` произойдет ошибка, то управление будет передано в блок `catch`. Блоков `catch` может быть несколько, например:

```

try
{
    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
catch (FormatException)
{
    Console.WriteLine("Вы ввели некорректное число {0}", inLine);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}

```

Здесь после блока `try` идут сразу два блока `catch` подряд, а если необходимо, то их может быть и больше, — каждый блок для разных классов исключительных ситуаций. После ключевого слова `catch` в скобках указывается имя класса исключительной ситуации, который мы хотим отлавливать. В первом случае `catch` будет обра-

батывать события класса `FormatException` и всех его наследников (если такие есть или если вы создали их).

Второй блок `catch` чуть интереснее, потому что в нем стоит класс `Exception`, который является базовым для всех, а значит, он отловит любое исключение, которое не было отловлено в других блоках. Помимо этого, в скобках указано имя переменной `e`, где нам передадут объект класса `Exception`, через который мы сможем получить информацию о произошедшей исключительной ситуации. В нашем примере выводится свойство `Message`, где находится описание ошибки. В первом блоке `catch` я не обращался к свойствам объекта, но если бы они были нужны мне, я мог бы объявить переменную, например, так:

```
catch (FormatException ef)
{
    ...
}
```

И все же указывайте переменную, только если она действительно необходима.

8.3. Оформление блоков *try*

Старайтесь включать в блок `try` только действительно необходимый код. Например, следующий код не слишком хорош:

```
while (true)
{
    try
    {
        Console.WriteLine("Введите число");

        string inLine = Console.ReadLine();
        if (inLine == "q")
            break;

        int i = Convert.ToInt32(inLine);
        Console.WriteLine("Вы ввели {0}", i);
    }
    catch (Exception e)
    {
    }
}
```

А плох он потому, что здесь в блок `try` заключено все содержимое цикла. Я размышлял как пессимист: а вдруг где-то произойдет ошибка, поэтому лучше ее заглушить. Поскольку мы не знаем, где во всем этом коде произойдет ошибка, то в блоке `catch` благодаря использованию класса `Exception` ловятся все события, но ничего не предложено в качестве какой-либо реакции на исключительную ситуацию. Такой подход является попыткой *заглушить* исключительную ситуацию без

попытки локализовать проблему. Никогда так не поступайте! Ошибки нужно исправлять, а не игнорировать.

Давайте разберемся, почему я в своих предыдущих примерах заключил в блок `try` строку вывода в консоль:

```
try
{
    int i = Convert.ToInt32(inLine);
    Console.WriteLine("Вы ввели {0}", i);
}
```

Да, эта строка не относится к проблемной ситуации, которую я пытаюсь локализовать с помощью `try`, но мне приходится так поступать, потому что переменная `i` объявлена внутри блока, и ее область видимости — только этот блок. За пределами блока я ее не увижу и не смогу вывести содержимое переменной в консоль. Для того чтобы вынести строку вывода переменной за пределы блока, мне нужно объявить переменную `i` перед блоком. Чтобы не делать этого, я пожертвовал переносом строки вывода в консоль внутрь блока `try`. Я надеюсь, любители качественного кода не покарают меня за этот ход.

Я люблю качественный код и стараюсь привить эту любовь и вам, но иногда отступаю от правил. Вы тоже можете отступать, но старайтесь делать это лишь тогда, когда без этого не обойтись.

Если вы хотите отлавливать все сообщения, и при этом вам не нужна переменная класса `Exception`, вы можете написать блок обработки следующим образом:

```
try
{
    ...
}
catch
{
}
```

Здесь после слова `catch` вообще нет указания на класс исключений. Это идентично написанию:

```
catch (Exception)
```

или:

```
catch (Exception e)
```

Такой метод применяют, когда не нужна переменная, а надо просто заглушить сообщение об ошибке.

8.4. Ошибки в визуальных приложениях

Если исключительная ситуация произошла в визуальном приложении, то .NET может отнести к такой ошибке не столь критично. Например, создайте окно и в нем поместите на форме поле ввода и кнопку, по нажатию на которую содержимое поля ввода будет переводиться в число:

```
int index = Convert.ToInt32(inputNumberTextBox.Text);
MessageBox.Show("Вы ввели: " + index);
```

Для конвертирования строки в число здесь также используется статичный метод `ToInt32()` класса `Convert`. Запустите приложение не в режиме отладки (не из среды Visual Studio) и попробуйте ввести в поле ввода что-то, не преобразуемое в число. В результате сработает исключительная ситуация. Для визуальных приложений .NET отображает другое окно, которое показано на рис. 8.3. Если есть возможность продолжить выполнение программы, то в окне будет кнопка **Продолжить**, по нажатию на которую можно попытаться продолжить выполнение программы. По нажатию кнопки **Сведения** пользователь может увидеть более подробную, но далеко не каждому понятную информацию об ошибке.

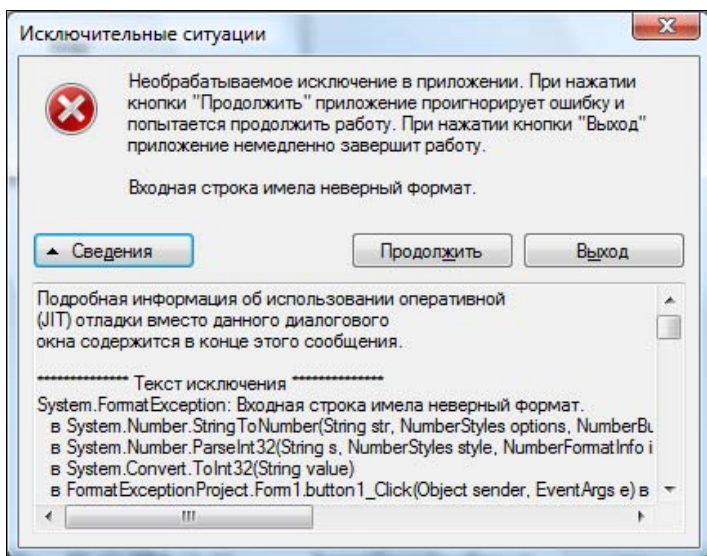


Рис. 8.3. Реакция на исключительную ситуацию в приложении WinForms

Но я бы не стал надеяться на .NET и на то, что программа сможет продолжить корректное выполнение. Намного эффективнее будет отловить исключительную ситуацию самостоятельно и предоставить пользователю более понятное сообщение об ошибке:

```
try
{
    int index = Convert.ToInt32(inputNumberTextBox.Text);
    MessageBox.Show("Вы ввели: " + index);
}
catch (FormatException)
{
    MessageBox.Show("Вы ввели некорректное число");
    return;
}
```

```
catch (Exception ex)
{
    MessageBox.Show("Неизвестная ошибка: " + ex.Message);
    return;
}
// другой код
...
```

В случае ошибки пользователь получит более простое и в то же время более понятное сообщение. При этом вы можете вызвать оператор `return`, чтобы прервать работу метода, или установить значение по умолчанию для переменной в блоке `catch` и продолжить выполнение. Это уже зависит от ваших предпочтений и от конкретной ситуации.

8.5. Генерирование исключительных ситуаций

Исключительные ситуации создаются не только системой. Вы можете самостоятельно создать исключительную ситуацию, и для этого служит ключевое слово `throw`. Где это можно использовать? Допустим, вам нужно, чтобы вводимое число было не более 10. С помощью исключительной ситуации эту проверку можно выполнить так:

```
if (index > 10)
    throw new Exception("Вы ввели слишком большое значение");
```

Ключевое слово `throw` генерирует исключение, объект которого мы создаем после указания этого слова. В нашем случае мы создаем экземпляр базового класса `Exception`. В качестве параметра конструктору класса мы передаем описание ошибки, которое попадет в свойство `Message` созданного нами объекта исключительной ситуации.

Конечно же, такой пример не очень нагляден, и в реальной жизни вы не будете производить проверки таким методом. Чтобы показать более наглядный пример, я решил вспомнить, как мы создавали индексатор для класса `Person`, через который обращались к объектам `Children` (см. главу 7). В нем не было никаких проверок, поэтому вы могли без проблем написать в коде что-то типа `person[-10]`, что привело бы к генерированию исключения выхода за пределы массива. Получается, что индексатор можно реализовать следующим образом (хотя этот способ и не всегда лучший, но он работает):

```
public Person this[int index]
{
    get
    {
        if (index >= Children.Count)
            throw
                new IndexOutOfRangeException("Слишком большое значение");
    }
}
```

```
if (index < 0)
    throw
        new IndexOutOfRangeException("Отрицательное запрещено");

return Children[index];
}
}
```

Теперь будет сгенерирована исключительная ситуация с более понятным описанием проблемы. Причем генерируется объект класса `IndexOutOfRangeException`, который как раз и проектировался специально для таких ситуаций, когда индекс выходит за границы.

Почему в этом коде нужно задействовать именно генерацию исключительной ситуации, а не просто отобразить диалоговое окно? Все очень просто — потому что этот класс может использоваться внешним классом, а он может быть где угодно и даже в консольном приложении, где наше диалоговое окно будет не совсем к столу. Внешний класс может не захотеть отображать никаких окон, а поставит с помощью блока `try...catch` простую заглушку, которая скроет наше сообщение и продолжит выполнение со значениями по умолчанию.

В таких случаях, когда проверка данных происходит внутри какого-то сервисного класса, а не класса конечного приложения или окна, лучше генерировать исключения, а не показывать какие-то свои окна. Если класс окна захочет, то отобразит нужное ему сообщение об ошибке, а если не захочет, то заглушит.

Если перед генерацией исключения нужно задать дополнительные параметры, то это можно сделать следующим образом:

```
IndexOutOfRangeException ex =
    new IndexOutOfRangeException("Ошибка");
ex.HelpLink = "http://www.flenov.info";
throw ex;
```

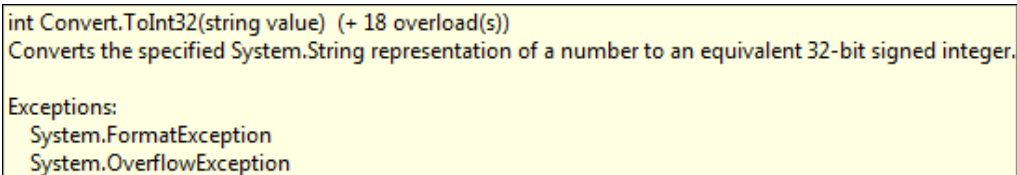
В этом примере я задаю у объекта дополнительное свойство `HelpLink` перед генерацией исключения.

8.6. Иерархия классов исключений

В .NET существует целая иерархия классов исключительных ситуаций. Вот основные ветки:

- ❑ `SystemException` — исключительные ситуации этого класса и его подклассов генерируются общезыковой средой выполнения CLR и являются исключениями системного уровня. Такие ошибки считаются неустраняемыми;
- ❑ `ApplicationException` — ошибки приложения. Если вы будете создавать свои классы исключительных ситуаций, то рекомендуется делать их потомками `ApplicationException`.

А как узнать, какие классы исключительных ситуаций может генерировать метод, чтобы знать, что обрабатывать? Эта информация находится вместе с описанием самих методов в MSDN. Но есть способ узнать классы быстрее — поставить курсор ввода на нужный метод и нажать комбинацию клавиш <Ctrl>+<K>+<I> или просто навести на метод указатель мыши. Должно появиться окно с кратким описанием метода и со списком возможных исключительных ситуаций во время вызова метода. На рис. 8.4 показано такое окно для метода `Convert.ToInt32()`. Как видите, в случае неудачного конвертирования метод может сгенерировать ошибку `FormatException` или `OverflowException` и оба класса из пространства имен `System`.



```
int Convert.ToInt32(string value) (+ 18 overload(s))
Converts the specified System.String representation of a number to an equivalent 32-bit signed integer.

Exceptions:
    System.FormatException
    System.OverflowException
```

Рис. 8.4. Краткая информация о методе `Convert.ToInt32()`

8.7. Собственный класс исключения

Вы можете создавать и свои классы исключительных ситуаций. Такое очень часто нужно делать, если объект, который создается при ошибке, должен содержать какие-то пользовательские данные. В качестве примера давайте создадим двигатель, который будет генерировать свой собственный класс исключительной ситуации при попытке запустить его, когда он уже работает. При этом объект исключения должен хранить ссылку на двигатель, сгенерировавший исключение. Пример такого класса двигателя можно увидеть в листинге 8.1.

Листинг 8.1. Пример класса двигателя

```
public class CarEngine
{
    public CarEngine(string name)
    {
        Working = false;
        Name = name;
    }

    public bool Working { get; private set; }
    public string Name { get; set; }
    public void StartEngine()
    {
        if (Working)
            throw new EngineException(this, "Двигатель уже работает");
    }
}
```

```
        Working = true;
    }

    public void StopEngine()
    {
        Working = false;
    }
}
```

В методе `StartEngine()` происходит проверка, работает ли уже двигатель, и если да, то выбрасывается исключение класса `EngineException`. Этому исключению передается текущий объект и сообщение об ошибке.

Теперь самое интересное — реализация класса `EngineException`. Ее можно увидеть в листинге 8.2.

Листинг 8.2. Реализация собственного класса исключения

```
public class EngineException: ApplicationException
{
    CarEngine engine;

    public EngineException(CarEngine engine, string message): base(message)
    {
        this.engine = engine;
    }

    public CarEngine Engine
    {
        get { return engine; }
    }
}
```

Здесь у нас объявлен класс `EngineException`, который является наследником класса `ApplicationException`. Обратите внимание на конструктор. Он получает в качестве параметров объект двигателя и сообщение. Объект двигателя сохраняется в переменной класса `EngineException`, и тут не возникает проблем. А вот описание нужно сохранить в свойстве `Message` предка `Exception`. Это не прямой предок (прямым является `ApplicationException`), а предок через поколение, т. е. предок предка.

Проблема заключается в том, что свойство `Message` класса `Exception` доступно нам лишь для чтения. Мы можем его изменить только в конструкторе класса при инициализации. А как вызвать конструктор предка? Если нужно вызвать конструктор этого же класса, только перегруженный, то после скобок с параметрами мы указываем двоеточие и обращаемся к конструктору через ключевое слово `this`. Если нужен конструктор предка, то вместо `this` надо поставить `base`. По количеству параметров, которые мы передадим `base`, платформа определит, какой из перегружен-

ных конструкторов предка мы хотим вызвать. Вот таким простым и хорошим способом мы перенаправили переменную `message` предку в конструктор, чтобы он сохранил значение в свойстве `Message`.

Есть еще один способ сделать так, чтобы свойство `Message` возвращало нужное нам значение, — переопределить свойство `Message`, как показано в листинге 8.3.

Листинг 8.3. Переопределение свойства предка

```
public class EngineException: ApplicationException
{
    CarEngine engine;
    String mymessage;

    public EngineException(CarEngine engine, string message)
    {
        this.engine = engine;
        this.mymessage = message;
    }

    public CarEngine Engine
    {
        get { return engine; }
    }

    public override string Message
    {
        get { return mymessage; }
    }
}
```

Переопределение свойства выгоднее тогда, когда вы хотите наделить свойство какой-то дополнительной функциональностью или дополнительными проверками. В нашем случае этого нет, поэтому можно оставить код, как в листинге 8.2.

Теперь запуск двигателя в коде может выглядеть следующим образом:

```
try
{
    engine.StartEngine();
}
catch (EngineException ee)
{
    MessageBox.Show("Двигатель '" + ee.Engine.Name +
        "'\nСгенерировал ошибку: '" + ee.Message + "'");
}
```

В этом случае программа работает синхронно, и мы можем узнать имя двигателя, сгенерировавшего исключение, просто обратившись к переменной `engine`. Со-

хранение объекта в исключении эффективно в тех случаях, когда работа с объектом идет асинхронно, т. е. когда ошибка появилась через некоторое время после вызова метода `StartEngine()`, а если в программе несколько двигателей, то без сохранения ссылки на объект в классе исключения узнать виновника будет труднее.

8.8. Блок *finally*

Блок `finally` удобен тем, что он выполняется в любом случае, вне зависимости от того, произошла исключительная ситуация или нет. Если `catch` выполняется только при ошибке, то `finally` отработает всегда:

```
try
{
    // код
}
finally
{
    // код выполнится вне зависимости от наличия исключения
}
```

Блок `finally` можно использовать совместно с блоком `catch`:

```
try
{
    // код
}
catch
{
    // произошла ошибка в коде
}
finally
{
    // код выполнится вне зависимости от наличия исключения
}
```

Эту связку удобно задействовать, когда вы работаете с какими-то выделяемыми ресурсами. Ярким примером такого ресурса могут быть файлы. Реальный код я сейчас не буду приводить, поэтому рассмотрим, как это может выглядеть:

```
try
{
    Открыть файл;
    Прочитать данные из файла;
    Обработать данные;
}
```

```
catch
{
    Сообщить пользователю об ошибке при работе с файлом;
}
finally
{
    if (файл открыт)
        Закрыть файл;
}
```

Некоторые программисты выносят операцию открытия файла из блока `try`, тогда в блоке `finally` не нужно проверять, открыт ли сейчас файл. Но методы открытия файлов тоже могут генерировать исключения, поэтому этот метод также нужно заключать в блок `try`. Я решил объединить все в одном блоке, хотя можно было и разделить на два.

Так как вызов закрытия файла написан в блоке `finally`, который выполняется вне зависимости от наличия исключения, то этим мы гарантируем, что файл будет закрыт в любом случае: обработали мы с ним корректно или нет.

8.9. Переполнение

Допустим, перед нами есть следующий код:

```
int x = 1000000;
int y = 3000;
int z = x * y;
```

По идее, в переменной `z` должен быть сохранен результат 3 000 000 000, но в реальности дело обстоит немного по-другому, — я бы сказал, совсем по-другому, потому что в результате перемножения мы получим $-1\,294\,967\,296$. Те, кто имеет опыт программирования, должны знать о проблеме переполнения, а остальным попробую объяснить. Дело в том, что тип данных `int` имеет границы данных от $-2\,147\,483\,648$ до $2\,147\,483\,647$. Результат перемножения `x * y` превышает максимально допустимое положительное число, поэтому произошло переполнение, и мы увидели некорректный результат.

В большинстве случаев числа `int` вполне достаточно, но если где-то нужно работать с большими числами, то переполнение может сыграть злую шутку. Самое страшное, что среда выполнения не сгенерирует никаких ошибок и мы не узнаем, что произошло это самое переполнение. Это сделано для того, чтобы вычисления проходили быстрее, а отслеживание вероятных выходов за предельные значения ложится на ваши плечи.

Если у вас есть код, который может выйти за пределы и повлиять на результат работы, то лучше заключить его в ключевое слово `checked`. В этом случае, если в указанных вычислениях произойдет переполнение, будет сгенерирована исключительная ситуация `OverflowException`. Например:

```

int x = 1000000;
int y = 3000;
try
{
    int z = checked(x * y);
    Console.WriteLine(z);
}
catch (OverflowException e)
{
    Console.WriteLine("Значение результата превышает пределы");
}
Console.ReadLine();

```

Если вам необходимо выполнить сразу несколько операторов, результат которых может привести к переполнению, то эти операторы могут быть заключены в фигурные скобки после слова `checked`:

```

checked
{
    Операторы;
}

```

Например, в следующем примере сразу три строки выполняются в блоке `checked`, и переполнение в любой из них приведет к генерации исключительной ситуации:

```

checked
{
    int z = x * y;
    z *= 10;
    x = z - x;
}

```

А если у вас программа использует большое количество чувствительных к результату вычислений с большими значениями — неужели придется везде ставить блоки `checked`? Во-первых, в этом случае лучше выбрать тип данных с более высоким значением — например, `Int64`. Во-вторых, генерация исключения переполнения отключена по умолчанию, но это можно изменить для каждого проекта в отдельности. Откройте окно свойств проекта (щелкнув правой кнопкой мыши на имени проекта в окне **Solution Explorer** и выбрав в контекстном меню **Properties**) и в разделе **Build** нажмите кнопку **Advanced**. В открывшемся окне **Advanced Build Settings** (рис. 8.5) поставьте флажок **Check for arithmetic overflow/underflow** (Проверять на арифметическое переполнение/потери значимости). Теперь исключительные ситуации будут генерироваться при любых переполнениях даже без использования ключевого слова `checked`.

Но тут же возникает другой вопрос: а что если мы включили генерацию исключительных ситуаций, но у нас в коде есть блок вычислений, который выполняется много раз, и он очень критичен ко времени выполнения? Например:

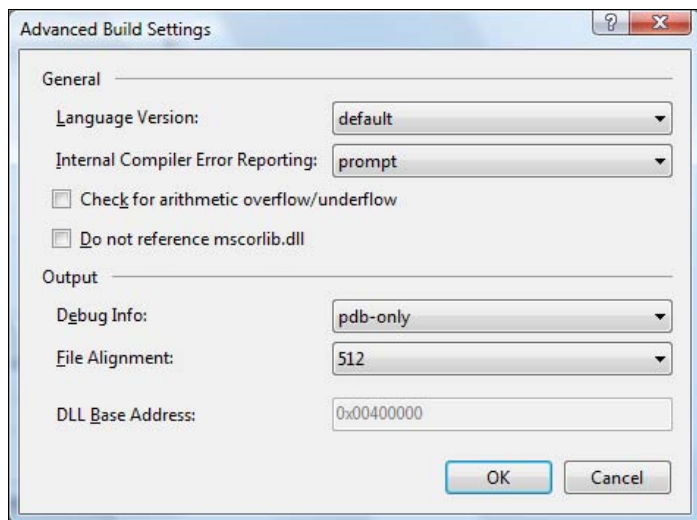


Рис. 8.5. Окно расширенных настроек проекта

```
for (int i = 0; i < 1000000; i++)  
{  
    Выполнить расчеты прогноза погоды;  
}
```

Этот цикл выполняется миллион раз подряд, а расчеты погоды достаточно сложные, но не всегда точные. Если где-то произойдет выход за пределы и вместо дождя мы предскажем солнце, то ничего страшного не произойдет. Подобные предсказания мы видим каждый день, особенно при долгосрочных прогнозах погоды, потому что прогнозирование — слишком неточная наука, и его точность зависит от срока.

Насчет «ничего страшного от выхода за пределы» я, конечно же, шучу. Если есть вероятность выхода за границы значений, то проверка необходима, и отключать ее не стоит, даже если это повысит скорость. Но допустим, что в расчете прогноза погоды мы используем самые большие переменные и гарантируем, что переполнения никогда не будет. Зачем миллион раз в каждой операции расчета проверять результат на выход за границы? Это плохо с точки зрения производительности, и отключение проверки может немного поднять скорость работы программы.

Если для всего проекта вы включили генерацию исключительных ситуаций при переполнении значения, то для определенного блока кода вы можете отключить проверку с помощью ключевого слова `unchecked`:

```
unchecked  
{  
    for (int i = 0; i < 1000000; i++)  
    {  
        Выполнить расчеты прогноза погоды;  
    }  
}
```

Любые вычисления в блоке `unchecked` не проверяются на выход за границы и не генерируют исключительных ситуаций. Несмотря на то что для всего проекта генерация исключений переполнения включена, в этом блоке при переполнении ничего не произойдет.

8.10. Замечание о производительности

Обработка исключительных ситуаций не отличается высокой производительностью, поэтому некоторые разработчики не рекомендуют ее к использованию и ищут свои методы отображения ошибок. Например, если пользователь ввел некорректные данные, то в коде чтения данных от пользователя можно сгенерировать исключение, а поймать его и отработать уже в другом месте. Это отличный и, на мой взгляд, правильный подход.

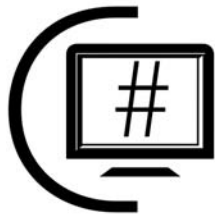
Иногда противники исключительных ситуаций, пытаясь решить проблему производительности, начинают использовать коды ошибок. Так, если метод чтения данных от пользователя столкнулся с проблемой, то вместо исключения этот метод может возвращать какой-нибудь магический код, который может сказать нам, удалось прочитать данные или нет.

Я же предпочитаю использовать именно исключительные ситуации, потому что они делают код лучше с точки зрения чтения и поддержки. Это удобнее, чем ситуации, когда какой-либо метод возвращает некий магический код ошибки, и мне потом приходится вспоминать, что он означает.

В `C#` строку можно превратить в число двумя методами: `Int32.Parse` или `Int32.TryParse`. Первый метод прост и удобен, но генерирует ошибку, если строку невозможно превратить в число, и по классу ошибки мы можем определить, что именно не так. Второй метод возвращает только `true` — в случае успеха или `false` — в случае ошибки. Этот метод работает быстрее, но проблема в том, что мы не знаем, что именно произошло, если невозможно превратить строку в число.

В тех частях приложения, где нужна высокая производительность, я использую `TryParse`, но если прямого и сильного требования к производительности нет, то обращаюсь к `Parse` и отлавливаю исключительные ситуации.

ГЛАВА 9



События

Для того чтобы создаваемые нами классы были действительно автономными и логически завершенными, иногда они должны не только выполняться самостоятельно, но и уметь сообщать о событиях, которые происходят внутри них. А внешние классы должны иметь возможность регистрироваться в качестве наблюдателей за определенными событиями.

В десктопном программировании *события* используются на каждом шагу. Если мы создадим окно, то оно может сообщать другим окнам о таких событиях, как изменение позиции или размера. В случае с кнопкой нам, конечно же, захочется узнать, когда пользователь щелкнул на ней.

В этой главе мы узнаем, что такое *делегаты*, и впервые заговорим о многопоточности, потому что делегаты являются одним из возможных способов вызова метода асинхронно.

9.1. Делегаты

Чтобы понять, что нам предстоит изучить в этой главе, мы должны разобраться, как происходит работа событий. Технически все очень просто. Допустим, мы хотим создать событие для нашего класса `Person`, которое будет вызываться, если у человека в классе `Person` изменилось имя или фамилия. У людей имя и фамилия меняются очень редко, и вполне логично задавать эти свойства в конструкторе класса и иметь возможность контролировать момент, когда в этих свойствах произошли изменения. Фамилия и имя могут использоваться для отображения человека в элементах управления визуального интерфейса, и эту информацию нужно обновлять.

Наш объект `Person`, генерирующий событие, мы назовем *издателем* события. Объекты, которые хотят получить событие, будут называться *подписчиками*. Подписчик должен сообщить издателю, что он хочет получать уведомления о возникновении какого-то события, причем на одно и то же событие могут подписаться несколько объектов разных классов. Чтобы издатель смог вызвать методы подписчиков, зарегистрированных на события, эти методы должны иметь строго опреде-

ленный формат для того или иного класса события, который описывается с помощью делегата.

Мы подошли к очень интересному понятию — *делегат*. Это тип, определяющий полную сигнатуру метода события, которая включает в себя тип возвращаемого значения и список параметров. Например, вот так описан в C# делегат, который имеет два параметра:

```
public delegate void EventHandler(Object sender, EventArgs e)
```

Здесь делегат описывает метод, который ничего не возвращает, и его два параметра это:

- `sender` — объект, который сгенерировал сообщение;
- `e` — объект класса `System.EventArgs`.

Такой делегат очень часто используется в десктопном программировании на языке C#. И хотя мы эту тему не рассматриваем, описание его все же очень интересно для понимания.

Если переменная описывает данные и их тип, то делегат описывает метод и говорит системе, какие параметры метод принимает и что возвращает.

Такое описание делегатов является необязательным, но желательным. Вы можете создать делегат, который не будет иметь параметров или будет содержать только один параметр.

9.2. События и их вызов

Делегаты описывают, как должен выглядеть метод в подписчике, который будет регистрироваться в качестве обработчика событий. Делегат также говорит издателю, метод какого типа будет вызываться и какие параметры нужно передать подписчику. Получается, что делегат является как бы договором между издателем и подписчиком на формат вызываемого метода.

Вы можете использовать делегаты одного класса в разных издателях и в разных событиях. Для объявления события определенного делегата служит ключевое слово `event`. Например, давайте добавим в наш класс `Person` свойство для хранения возраста, и при попытке изменить возраст будет вызываться событие. Нам просто нужно проинформировать классы-подписчики о том, что изменился возраст, поэтому можно не создавать собственный делегат, а использовать готовый: `EventHandler`.

Итак, с помощью ключевого слова `event` объявляем событие с именем `AgeChanged` класса `EventHandler`:

```
public event EventHandler AgeChanged;
```

Событие объявляется публичным (`public`), чтобы его могли отлавливать сторонние подписчики (подписчики любого класса).

Чтобы сгенерировать событие, нужно лишь вызвать его как простой метод, например:

```
AgeChanged(this, new EventArgs());
```

Событие `AgeChanged` у нас объявлено как делегат `EventHandler`. Этот делегат получает в качестве параметров объект, который сгенерировал событие, и пустой экземпляр класса `EventArgs`. Чтобы передать объект, мы просто передаем в первом параметре `this`, а во втором параметре создаем экземпляр класса `EventArgs`.

Такой вызов метода пройдет без ошибок только в том случае, если есть хотя бы один подписчик для нашего события. Если подписчиков нет, то переменная события `AgeChanged` будет равна нулю, и этот вызов сгенерирует исключительную ситуацию. Как поступить в таком случае? Нет, отлавливать исключительную ситуацию будет не очень хорошим решением. Намного лучше просто проверить событие на равенство нулю:

```
if (AgeChanged != null)
    AgeChanged(this, new EventArgs());
```

Этот код уже более корректен, потому что перед генерацией события он проверяет `AgeChanged` на равенство нулю. Если событие не равно нулю, то существует хотя бы один обработчик и мы можем генерировать событие.

Событие может вызываться только в том классе, в котором оно объявлено. Это значит, что вы не можете сгенерировать событие `AgeChanged` из объекта класса `Zarplata`.

Теперь посмотрим, как может выглядеть свойство `Age` для хранения возраста:

```
int age = 0;
public int Age
{
    get { return age; }
    set
    {
        if (value < 0)
            throw new Exception("Возраст не может быть отрицательным");

        age = value;

        if (AgeChanged != null)
            AgeChanged(this, new EventArgs());
    }
}
```

Я специально выбрал такое свойство, потому что оно интересно еще и с точки зрения ограничения. Возраст не может быть отрицательным, поэтому мы должны обязательно добавить в класс проверку на попытку установить отрицательный возраст. Но что делать, если произошла такая попытка? Можно просто проигнорировать ее и не изменять значение, но, на мой взгляд, более корректным решением было бы сгенерировать исключительную ситуацию, чтобы проинформировать о проблеме.

Иначе пользователь будет думать, что он изменил возраст, хотя на самом деле класс проигнорировал отрицательное значение. Именно это и происходит в аксессоре `set` в самом начале. Если проверка прошла успешно, то мы изменяем значение и генерируем событие.

Как теперь использовать событие в подписчике? Для визуальных компонентов в окне свойств появляется вкладка **Events**, где мы можем создавать обработчики событий. Но перед нами не визуальный компонент, и как поступить в этом случае? Придется писать код регистрации объекта в качестве подписчика вручную.

Создайте новое визуальное приложение `WinForms` и поместите на форму компоненты, с помощью которых можно будет работать с объектом класса `Person`. В общем-то, нам достаточно поля ввода `NumericUpDown` для ввода возраста и кнопки, по нажатию на которую станет изменяться значение возраста в объекте `p` класса `Person`:

```
p.Age = (int)ageNumericUpDown.Value;
```

В листинге 9.1 вы можете увидеть полный код класса такой формы.

Листинг 9.1. Класс формы с ручной регистрацией события

```
public partial class Form1 : Form
{
    Person p = new Person("Алексей", "Иванов");

    public Form1()
    {
        InitializeComponent();

        ageNumericUpDown.Value = p.Age;
        // регистрация события
        p.AgeChanged += new EventHandler(AgeChanged);
    }

    public void AgeChanged(Object sender, EventArgs args)
    {
        Person p = (Person)sender;
        MessageBox.Show("Возраст изменился на " + p.Age.ToString());
    }

    private void ageChangedButton_Click(object sender, EventArgs e)
    {
        p.Age = (int)ageNumericUpDown.Value;
    }
}
```

В этом примере в классе формы `Form1` объявляется объектная переменная `p` класса `Person`, с которой мы и будем работать. По нажатию кнопки у этого объекта изменяется возраст. Самое интересное происходит в конструкторе:

```
p.AgeChanged += new EventHandler(AgeChanged);
```

Что такое `AgeChanged`? Это событие, которое мы описывали в начале главы, и оно является типом делегата `EventHandler`. В событии регистрируются подписчики. Чтобы добавить свой объект в качестве получателя события, нужно выполнить операцию добавления к текущему значению нового экземпляра обработчика с помощью операции `+=`. Если нужно удалить обработчик события, то необходимо выполнить операцию `-=`.

Что мы прибавляем к событию? А прибавляем мы экземпляр делегата, которым является наше событие. Наше событие является делегатом `EventHandler`, а значит, мы должны добавить экземпляр `EventHandler`. Как создать экземпляр делегата? Такой вопрос еще интереснее, потому что в качестве параметра экземпляры делегатов получают имя метода, который должен вызываться в ответ на событие. Этот метод должен иметь точно такие же параметры, что и в описании делегата. В нашем случае мы передаем делегату `EventHandler` метод `AgeChanged()`. Именно этот метод станет вызываться каждый раз, когда объект `Person` будет генерировать событие.

На первый взгляд все выглядит немного запутанно и сложно, но это только на первый взгляд, и все не так уж страшно на практике. Если что-то оказалось непонятным из описания, то я надеюсь, что пример расставит все на свои места. В следующем разделе мы научимся описывать собственного делегата и еще немного закрепим эту тему.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter9\EventHandlerProject` сопровождающего книгу электронного архива (см. приложение).

9.3. Использование собственных делегатов

Если мы хотим не просто проинформировать объект о наступлении события, но и передать обработчику дополнительную информацию, то нужно расширить делегат `EventHandler` собственной реализацией. Допустим, нам нужно сделать так, чтобы при изменении фамилии или имени вызывался метод события, в который передавалось бы новое значение, и само событие вызывалось бы до изменения, чтобы класс-подписчик мог отменить изменение.

Итак, нам нужен такой делегат для нашего события, который в качестве второго параметра будет передавать объект класса, содержащий новое значение имени/фамилии и какой-то индикатор. Индикатор и определит, надо принять изменение или нет.

Почему этот класс нужно передавать именно во втором параметре делегата и почему не надо передавать текущее значение, а только новое? Ответ кроется в первом параметре. Как мы уже говорили, хорошее событие должно передавать в первом параметре объект, который сгенерировал событие. Получается, что первое место уже занято. Так как в этом параметре находится нужный нам объект и событие вызвано до изменений, то мы можем получить текущее значение через объект из первого параметра.

Соответственно нам нужен такой делегат, который в первом параметре передает объект, сгенерировавший событие, а во втором параметре — наш класс, который является расширением базового `EventArgs`. Для начала посмотрим на класс-расширение базового. Его возможная реализация приведена в листинге 9.2.

Листинг 9.2. Реализация собственного класса для параметров сообщений

```
public class NameChangedEventArgs : EventArgs
{
    // перечисление, определяющее тип изменения
    public enum NameChangingKind { FirstName, LastName }

    // конструктор
    public NameChangedEventArgs(string newName, NameChangingKind nameKind)
    {
        NewName = newName;
        NameKind = nameKind;
        Canceled = false;
    }

    public string NewName { get; set; }
    public bool Canceled { get; set; }

    public NameChangingKind NameKind { get; set; }
}
```

Наш класс является наследником от базового `EventArgs` и имеет три дополнительных свойства:

- ❑ `NewName` — строка, в которой хранится новое имя, которое мы хотим установить;
- ❑ `Canceled` — если это свойство равно `true`, то изменения нельзя принимать;
- ❑ `NameKind` — тип изменяемого имени. Этот параметр является перечислением `NameChangingKind`, которое объявлено тут же в классе, и позволяет определить, что изменяется: имя или фамилия.

Все три свойства задаются в конструкторе, но если первое и третье передаются в конструктор в качестве параметра, то свойство `Canceled` просто устанавливается в `false`, т. е. по умолчанию изменения должны быть приняты.

Теперь посмотрим, как можно объявить делегат и использовать его в нашем классе `Person` (листинг 9.3).

Листинг 9.3. Использование делегата

```
public class Person : IEnumerable
{
    // делегат
    public delegate void NameChanged(Object sender, NameChangedEventArgs args);
```

```
// объявление событий
public event NameChanged FirstNameChanged;
public event NameChanged LastNameChanged;

...

// свойство имени
string firstName;
public string FirstName
{
    get { return firstName; }
    set
    {
        if (FirstNameChanged != null)
            FirstNameChanged(
                this,
                new NameChangedEvent(value,
                    NameChangedEvent.NameChangingKind.FirstName)
            );
        firstName = value;
    }
}

// свойство фамилии
string lastName;
public string LastName
{
    get { return lastName; }
    set
    {
        if (LastNameChanged != null)
        {
            NameChangedEvent changeevent = new NameChangedEvent(value,
                NameChangedEvent.NameChangingKind.FirstName);
            LastNameChanged(this, changeevent);
            if (changeevent.Canceled)
                return;
        }

        lastName = value;
    }
}

...
}
```

В новом варианте класса `Person` в самом начале объявляется делегат с именем `NameChanged`. Имя может быть любым, потому что оно нужно только для удобства

использования, — желательно, чтобы имя отражало суть делегата. После этого объявляются два события: `FirstNameChanged` и `LastNameChanged`, которые имеют формат делегата `NameChanged`.

ПРИМЕЧАНИЕ

Некоторое количество кода из листинга 9.3 удалено в целях экономии места — полный вариант исходного кода примера к этому разделу можно найти в папке *Source\Chapter9\OwnDelegate* сопровождающего книгу электронного архива (см. приложение).

Следующим интересным местом в коде является аксессор `set` свойства `FirstName`, где мы должны сгенерировать событие. После проверки события на неравенство нулю генерируем событие:

```
FirstNameChanged(this,
    new NameChangedEvent(value,
        NameChangedEvent.NameChangingKind.FirstName)
);
```

Событию передаются два параметра в соответствии с форматом делегата: ссылка на текущий объект и экземпляр класса `NameChangedEvent`. Этот экземпляр создается непосредственно в месте передачи параметра, а конструктору передается новое значение имени и соответствующее значение перечисления.

В нашем случае я просто проигнорировал возможность отмены для этого свойства, т. е. отменить изменение имени внешнему классу не удастся. Это сделано намеренно, а вот у свойства `LastName` генерация события в аксессоре `set` выглядит немного по-другому:

```
NameChangedEvent changeevent = new NameChangedEvent(value,
    NameChangedEvent.NameChangingKind.FirstName);

LastNameChanged(this, changeevent);
if (changeevent.Canceled)
    return;
```

В этом случае экземпляр класса `NameChangedEvent` создается явно и сохраняется в переменной `changeevent` класса `NameChangedEvent`. Это необходимо, чтобы после генерации события мы смогли обратиться к объекту и узнать значение свойства `Canceled` — не изменилось ли оно в процессе обработки сообщения подписчиками. Если оно изменилось на `true`, то дальнейшее выполнение аксессора прерывается.

Теперь посмотрим, как это может использоваться нами в коде внешнего класса. Для этого нам понадобится программа наподобие той, которую мы написали в разд. 9.1, только в этом случае она должна изменять имя и фамилию.

Следующие две строки нужно добавить в конструктор формы, чтобы подписаться на обработку событий изменения имени и фамилии:

```
p.FirstNameChanged += new Person.NameChanged(FirstNameChanged);
p.LastNameChanged += new Person.NameChanged(LastNameChanged);
```


Так как события `FirstNameChanged` и `LastNameChanged` являются делегатами типа `NameChanged`, то и добавлять к ним нужно методы именно такого типа. Метод `FirstNameChanged()` вы можете увидеть в исходном коде в электронном архиве, и в нем я просто вывожу сообщение о том, что имя изменилось, а вот обработчик события изменения фамилии выглядит немного интереснее:

```
public void LastNameChanged(Object sender, NameChangedEventArgs args)
{
    Person p = (Person)sender;
    if (MessageBox.Show("Попытка изменить фамилию " + p.LastName +
        " на " + args.NewName, "Внимание",
        MessageBoxButtons.OKCancel) == DialogResult.Cancel)
        args.Canceled = true;
}
```

В обработчике события с помощью статического метода `Show()` класса `MessageBox` отображается диалоговое окно. Причем я задал такой вариант конструктора, который принимает три параметра: текст, заголовок и кнопки, а возвращает результат, который выбрал пользователь. В качестве кнопок я выбрал тип `MessageBoxButtons.OKCancel`, чтобы отобразить кнопки **Да** и **Отмена**. Если пользователь выберет **Да**, то результатом работы метода будет `DialogResult.OK`, иначе — `DialogResult.Cancel`. Я проверяю результат на равенство второму значению и, если пользователь выбрал отмену, изменяю свойство `Canceled` объекта `args`, т. е. отменяю изменение фамилии.

Попробуйте запустить пример и протестировать его в действии.

На самом деле, когда подписчик собирается зарегистрироваться в качестве обработчика событий, он не обязан использовать полный формат. *Полным форматом* называется способ, когда вы добавляете результат создания нового экземпляра обработчика события, — например, как мы это делали ранее:

```
p.FirstNameChanged += new Person.NameChanged(FirstNameChanged);
```

Я привык писать полный вариант, но можно обойтись и более короткой формой записи, просто добавив обработчику события только имя метода:

```
p.FirstNameChanged += FirstNameChanged;
```

При этом метод `FirstNameChanged()` должен соответствовать делегату, который используется событием, т. е. должен принимать точно определенные параметры и возвращать значение, определенное при объявлении делегата.

Какой метод подписки на события выберете вы, зависит от ваших личных предпочтений.

9.4. Делегаты изнутри

Когда вы объявляете делегат, компилятор создает в коде изолированный класс для него, который будет наследником класса `MulticastDelegate`, а `MulticastDelegate`, в свою очередь, является наследником `Delegate` — базового класса для делегатов.

Оба эти класса — системные, и вы не можете создавать собственных наследников, да я и не вижу необходимости в таком наследовании.

Классы-предки делегатов реализуют методы, необходимые событию для того, чтобы хранить список методов вызова. Когда подписчик подписывается на событие с помощью операции `+=`, то вызывается метод `Combine()`. Когда подписчик отписывается от события с помощью операции `-=`, вызывается метод `Remove()` класса `Delegate`.

Для каждого класса делегата, помимо наследуемых от `MulticastDelegate` и `Delegate` методов, система добавляет еще два специализированных метода: `BeginInvoke()` и `EndInvoke()`. Есть еще один очень важный метод: `Invoke()`, с которого мы и начнем рассмотрение делегата.

Метод `Invoke()` используется для генерации события синхронно. Синхронный вызов заставляет издателя ждать, пока подписчики не обработают событие, и только после этого издатель продолжает работу. До сих пор мы использовали именно синхронный вызов, хотя напрямую не вызывали метода `Invoke()`. Просто, если явно не указан метод, используется именно `Invoke()`. То есть синхронный вызов можно было бы сделать и так:

```
AgeChanged.Invoke(this, new EventArgs());
```

Методы `BeginInvoke()` и `EndInvoke()` позволяют генерировать событие асинхронно. В этом случае объект-издатель создает отдельный поток, внутри которого и происходит вызов методов подписчиков, а сам в это время продолжает выполняться параллельно в своем потоке. Это значит, что обработчики событий будут выполняться параллельно с работой основного объекта. Это хорошо, а иногда просто необходимо, но у асинхронного вызова есть свои нюансы.

Давайте вспомним пример, в котором издатель генерирует сообщение при попытке изменения фамилии:

```
LastNameChanged(this, changeevent);  
if (changeevent.Canceled)  
    return;
```

Что произойдет, если это событие будет сгенерировано асинхронно? Свойство `Canceled`, скорее всего, всегда будет равно `false`, потому что при генерации события не произойдет блокировка выполнения потока команд. Выполнение будет продолжаться параллельно с работой подписчиков. Так как проверка свойства `Canceled` происходит сразу после генерации события, я думаю, что ни один подписчик не успеет изменить свойство, работая параллельно с проверкой:

```
if (changeevent.Canceled)  
    return;
```

Так что не пытайтесь получать какие-то данные от подписчиков, работая в асинхронном режиме, без синхронизации выполняемых потоков. Если нужен результат, проще использовать синхронный вызов. К асинхронному вызову лучше прибегать только тогда, когда он действительно необходим и приносит пользу. Иначе лучше ограничиться синхронным вариантом. Более подробно об этом мы поговорим в разд. 15.3.

9.5. Анонимные методы

Когда метод обработчика события выполняет несколько операций, то создавать ради этого полноценный метод вполне резонно. А если нужно выполнить только одну операцию, то писать такое количество кода достаточно проблематично и скучно. Но для решения этой проблемы в .NET есть один интересный способ сокращения труда — *анонимные методы*. Например, в листинге 9.4 показана обработка события изменения возраста с использованием анонимного метода. Точно такой же код, но без анонимности, мы использовали в листинге 9.1.

Листинг 9.4. Анонимный метод для обработки события

```
public Form1()
{
    InitializeComponent();

    // читаем свойства объекта Person
    firstNameTextBox.Text = p.FirstName;
    lastNameTextBox.Text = p.LastName;
    ageNumericUpDown.Value = p.Age;

    // обработчику события присваивается код анонимного метода
    p.AgeChanged += delegate(Object sender, EventArgs args)
    {
        Person person = (Person)sender;
        MessageBox.Show("Возраст изменился на " + person.Age.ToString());
    };
}
```

В листинге 9.4 нет никаких реальных методов для обработки события. Вместо этого событию `AgeChanged` прибавляется следующая конструкция:

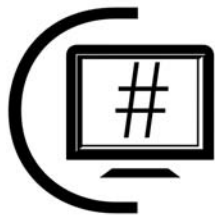
```
delegate(Object sender, EventArgs args)
{
    Person person = (Person)sender;
    MessageBox.Show("Возраст изменился на " + person.Age.ToString());
};
```

После ключевого слова `delegate` в круглых скобках записаны параметры, которые должны передаваться обработчику события. После этого в фигурных скобках идет код, который и станет выполняться при возникновении события. Код не будет выполнен во время работы конструктора, а только при возникновении события. Такое объявление кода и называется *анонимным*, потому что реального объявления метода нет, и у кода нет имени (поэтому он и анонимный), как у метода. Мы просто сообщаем непосредственно событию операторы, которые нужно выполнять в ответ на это событие.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter9\Anonym` сопровождающего книгу электронного архива (см. приложение).

ГЛАВА 10



LINQ

LINQ (Language Integrated Query, интегрированные в язык запросы) — это очень удобный метод доступа к данным массивов, появившийся в .NET 3.5. С помощью LINQ можно писать запросы в стиле SQL или вызывать методы C# и работать не только с массивами в памяти, но и с XML-файлами и даже с базами данных.

Я использую LINQ для работы с массивами и XML-файлами, но не очень хорошо отношусь к подобному методу доступа к базам данных.

10.1. LINQ при работе с массивами

Доступ к массивам с помощью запросов называют *Linq to Objects*, и он будет работать на любых массивах, которые реализуют интерфейс `IEnumerable`. Для того чтобы получить доступ к возможностям LINQ, необходимо подключить пространство имен `System.Linq`.

Допустим, у нас есть массив `people` типа `List<Person>`, а типизированные массивы как раз реализуют нужный нам интерфейс.

Те задачи, которые решаются с помощью LINQ, можно решить и самому посредством простых циклов по всем элементам массива. Но если привыкнуть к языку запросов, то он будет выглядеть в коде намного компактнее, красивее и удобнее для чтения.

Первое, что нужно сделать, — это, как уже отмечалось, подключить пространство имен `System.Linq`. В Microsoft явно очень сильно любят LINQ, и в шаблоне Visual Studio для создания новых файлов уже прописано подключение этого пространства имен по умолчанию, так что в большинстве случаев вам не нужно будет об этом заботиться.

10.1.1. SQL-стиль использования LINQ

Теперь рассмотрим пример с массивом людей, в котором мы хотим найти всех детей не старше 16 лет. На языке запросов в коде это будет выглядеть так:

```
List<Person> people = GetPeople()
var results = from p in people
               where p.Age < 16
               select new {p.FirstName, p.LastName, p.Age};
```

Синтаксис LINQ очень похож на SQL (язык запросов, который используется для доступа к базам данных). Я покажу вам, как я читаю подобные запросы, и надеюсь, что мой метод поможет вам.

Запросы начинаются с написания секции `from`, которая говорит, откуда мы должны получать данные:

```
from p in people
```

Эту строку нужно читать так: «для каждого элемента `p` из массива `People`». В нашем случае `p` становится переменной массива, которая как бы станет хранить элементы массива при обработке, а мы через нее сможем обращаться к элементам `people`. При работе с массивами мы привыкли работать с индексами и часто помечали их буквой `i`, а здесь мы работаем с объектами, и каждый элемент будет помещен в виртуальную переменную `p`, как в случае с циклами `foreach`.

Читаем дальше: «найти элементы, где у `p` свойство `Age` меньше 16»:

```
where p.Age < 16
```

И затем: «в качестве результата создать новый анонимный объект из трех свойств: имя, фамилия и возраст»:

```
select new { p.FirstName, p.LastName, p.Age }
```

Я в ИТ уже давно, но только однажды в 2021 году работал в течение 9 месяцев в компании, где использовали подобный подход для выполнения LINQ-запросов.

Но прежде, чем мы рассмотрим второй метод, я хотел бы вернуться к конструкции:

```
new { p.FirstName, p.LastName, p.Age }
```

Здесь мы создаем анонимный тип — класс без имени типа данных. Как называть переменную, в которую будет записан результат? Какой тип ей указывать?

Получается, что результатом запроса к объектам будет массив из объектов класса, тип которого мы не можем предсказать, и мы также не знаем его имени, чтобы можно было обращаться к нему в коде? Да, именно так. Это анонимные классы, которые мы рассматривали ранее, и именно поэтому результат запроса я записываю в переменную типа `var`. Необходимость использовать `var` в таких случаях наступает, потому что мы просто не знаем типа данных, а точнее — он анонимный, без имени.

Можно создавать и типизированные объекты, но именно с LINQ в основном используют анонимные. Дело в том, что при частом выполнении LINQ-запросов объявление для каждого из них отдельного класса может быть накладно. Поэтому мы просто создаем анонимный класс и используем его.

Чтобы отобразить результат, можно просто перебрать элементы массива:

```
foreach (var p in results)
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
```

Компилятор достаточно умный, чтобы увидеть, что у массива анонимных объектов есть свойства имени и фамилии.

Следующий пример показывает, как можно получить доступ к свойству `FirstName`, используя возможности *рефлектора*. Для использования этого кода нужно подключить пространство имен `System.Reflection`:

```
foreach (var person in results) {
    Type anonymousType = person.GetType();
    PropertyInfo pi = anonymousType.GetProperty("FirstName");
    return (string)pi.GetValue(person, null);
}
```

Здесь запускается цикл перебора всех объектов в массиве, потому что запрос вернет нам именно массив. Для каждого анонимного объекта в массиве мы выполняем три действия:

- ❑ сначала нужно получить тип нашего анонимного объекта, а это делается с помощью метода `GetType`. Этот метод наследуется от класса `Object`. А т. к. в C# все классы в качестве самого первого предка имеют этот класс, то и метод `GetType` будет абсолютно у любого объекта C#. Результатом выполнения метода является объект класса `Type`, который знает все о нашем объекте;
- ❑ но нам нужен не объект, а свойство. А информация о свойстве находится в объектах класса `PropertyInfo`, и эту информацию можно получить с помощью вызова метода `GetProperty`;
- ❑ а вот уже используя этот объект, можно получить непосредственно значение с помощью метода `GetValue`.

Впрочем, вы не обязаны работать с анонимными объектами — вполне реально работать и с простыми. Вот так мы можем с помощью запроса вернуть массив из объектов `Person`:

```
var results = from p in people
    where p.Age < 16
    select new Person
    {
        FirstName = p.FirstName,
        LastName = p.LastName,
        Age = p.Age;
    };
```

Я показал анонимные объекты вместе с LINQ, потому что как раз с ним я вижу их чаще всего. Сам же я по возможности стараюсь объявлять все классы, которые мне нужны явно.

10.1.2. Использование LINQ через методы

Я предпочитаю использовать методы, а не запросы, потому что они более привычны и их проще читать. Как только вы подключили пространство имен `System.Linq`, у всех коллекций, реализующих `IEnumerable`, появляется набор методов для выполнения запросов.

Чаще всего мне приходится работать с методом `Where`, который позволяет искать нужные объекты массива. В операторе `Where` вы можете писать практически любой .NET-код, манипулировать данными и производить любые проверки.

Например, следующая строка решает ту же задачу, что и пример из *разд. 10.1.1*, — ищет всех, кому нет 16 лет:

```
IEnumerable<Person> underage = people.Where (r => r.Age < 16);
```

В качестве параметра метод `Where` принимает специальное выражение. В нем буква `r` — это просто псевдоним для переменной, которая будет олицетворять каждую строку из массива `people`. С тем же успехом можно было взять и любую другую букву алфавита, и я часто вижу, что программисты берут букву `m`. Я же решил использовать `r`, потому что это первая буква в слове `row`.

Итак, слева от `=>` указывается имя переменной (псевдоним), а справа можно обращаться к свойствам объектов строк. Я все так же использую конструкцию:

```
r.Age < 16
```

чтобы указать, что я ищу в массиве всех людей, которые моложе 16 лет.

Результатом выполнения этого метода будет массив `IEnumerable`. Если вы больше предпочитаете работать с богатыми возможностями списков и если они действительно вам нужны, то результат достаточно легко привести к списку, если вызвать `ToList()`:

```
List<Person> underage = people.Where (r => r.Age < 16).ToList();
```

Просто так приводить к списку не стоит, потому что тут есть очень серьезная разница в результате, и мы рассмотрим ее в *разд. 10.2*.

Работая с методами, мы также можем создавать анонимные объекты, и для этого используется метод `Select`:

```
var underage = people.Where(r => r.Age < 16)
    .Select(m => new { FirstName = m.FirstName, LastName = m.LastName });
foreach (var p in underage)
    Console.WriteLine(p.FirstName + " " + p.LastName);
```

10.2. Магия *IEnumerable*

Все методы LINQ, возвращающие более одного элемента, возвращают тип `IEnumerable`. Глядя на первую букву имени типа, можно догадаться, что это интерфейс. Так как экземпляры интерфейсов создавать нельзя, значит, в реальности мы получаем какой-то объект какого-то класса, который реализует `IEnumerable`.

Прежде чем я перейду к конкретному примеру, нам нужно задаться еще одним вопросом: когда система производит фильтрацию данных? В момент, когда мы вызываем метод `Where`? Нет, это происходит, когда мы обращаемся к элементам массива или начинаем их перебирать.

Вот теперь давайте посмотрим на пример, показанный в листинге 10.1.

Листинг 10.1. Пример множественного использования `IEnumerable`

```
List<Person> people = SampleHelper.CreatePersons();

IEnumerable<Person> results = people.Where(r => r.Age < 16);

foreach (var p in results) {
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
}

Console.WriteLine("Добавим значение:");
people.Add(new Person() { FirstName = "Михаил", LastName = "Сепреев", Age = 10 });

foreach (var p in results) {
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
}
```

В этом примере создается список людей типа `List<Person>`.

ПРИМЕЧАНИЕ

Исходный код метода `SampleHelper.CreatePersons` можно найти в папке *Source\Chapter10\InqObjects* сопровождающего книгу электронного архива (см. приложение).

Метод `SampleHelper.CreatePersons` всего лишь создает список и наполняет его несколькими значениями, которые мы можем использовать для наших тестов.

После этого мы ищем людей, кому не исполнилось 16 лет, с помощью LINQ-метода `Where`. В этот момент реальной фильтрации не происходит — нам только возвращают реализующий интерфейс `IEnumerable` объект, который умеет фильтровать данные в соответствии с указанными нами условиями.

Теперь мы запускаем цикл `foreach`, внутри которого отображаются найденные люди. В моем примере в списке будет присутствовать один человек, и вы должны будете увидеть его на экране.

Затем в список `people` добавляется еще один человек, возраст которого явно меньше 16 лет. Я не выполняю больше никакого метода `Where`, не получаю новый объект из списка `people`, а просто запускаю перебор на результате `result`, который был уже получен ранее. И если посмотреть результат, то теперь мы увидим двух человек младше 16.

Мы начали новый цикл `foreach` на объекте результата, а интерфейс `IEnumerable` начал перечислять все с начала, и уже существующий объект отфильтровал данные заново.

Это и преимущество, и недостаток одновременно. Каждый раз, когда вы обращаетесь к переменной `result` и начинаете перечисление сначала, система вынуждена фильтровать данные, а это дополнительная нагрузка на ресурсы процессора. Если вы знаете, что данные между двумя обращениями к результату не будут меняться, то лучше привести результат в более простой формат — например: `List<Person>`.

Посмотрим на листинг 10.2. Здесь выполняется примерно тот же код, разница лишь в том, что во второй строке кода после вызова `Where` происходит приведение к списку — идет вызов `ToList()`. Именно в этот момент произойдет фильтрация, и теперь уже повторное обращение к переменной после изменения оригинального массива не увидит изменений.

Листинг 10.2. Пример приведения `IEnumerable` к списку

```
List<Person> people = SampleHelper.CreatePersons();

IEnumerable<Person> results = people.Where(r => r.Age < 16).ToList();

foreach (var p in results) {
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
}

Console.WriteLine("Добавим значение:");
people.Add(new Person() { FirstName = "Михаил", LastName = "Сергеев", Age = 10 });

foreach (var p in results) {
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
}
```

Можно создавать цепочки из фильтров, и они не будут отрабатываться, пока вы не обратитесь к переменной результата. Посмотрим на следующий пример:

```
List<Person> people = SampleHelper.CreatePersons();
IEnumerable<Person> byAge = people.Where(r => r.Age < 16);
IEnumerable<Person> byCityAndAge = byAge.Where(r => r.City == "Постов");

foreach (var p in byCityAndAge) {
    Console.WriteLine(p.FirstName + " " + p.LastName + ": " + p.Age);
}
```

В этом примере у нас снова создается массив из объектов, описывающих данные людей. Во второй строке с помощью LINQ я фильтрую по возрасту, а в третьей строке уже отфильтрованный по возрасту список фильтруется еще и по городу.

Когда мы обращаемся к переменной `byCityAndAge`, то платформа видит, что переменная использует другой объект `IEnumerable`, и будут оценены оба сразу. Не каждый в отдельности, а оба одновременно.

Рассмотрим случай приведения кода к списку:

```
List<Person> people = SampleHelper.CreatePersons();  
List<Person> byAge = people.Where(r => r.Age < 16).ToList();  
List<Person> byCityAndAge = byAge.Where(r => r.City == "Ростов").ToList();
```

Здесь после каждой фильтрации результат приводится к списку, а значит, фильтрация будет происходить дважды.

Таким образом вы можете строить целые цепочки фильтров, а платформа оптимизирует их и выполнит за минимальное количество шагов. В худшем случае я видел только два шага, когда первый выполнялся базой данных, а потом выполнялся еще один на стороне .NET, если фильтр был настолько сложный, что его невозможно было привести в SQL-запросе. Но это случалось при использовании LINQ для доступа к базе данных, а в этой главе мы говорим про массивы.

Рассмотрим такой гипотетический пример:

```
// получить адреса из базы данных  
IEnumerable<Person> people = db.Where(/* фильтр */);  
// Получить адреса из базы данных  
IEnumerable<PersonAddress> address =  
    db.Where(m => people.Contains(p => p.PersonID));  
foreach (var p in people) {  
    var personAddress = address.Where(m => m.PersonID = a.PersonID);  
    // выполняем какие-то действия над personAddress  
}
```

Сразу скажу, что с точки зрения производительности этот пример неэффективный, и причина не только в `IEnumerable`, который я пытаюсь показать, но и в самой логике. Для подобных примеров лучше использовать хеш-таблицы или словари, но это уже тема отдельной книги по алгоритмам.

Здесь у нас есть два больших списка: люди и адреса, которые получаются из базы данных с помощью LINQ. Мы LINQ для доступа к базам не рассматриваем, поэтому просто представим, что может существовать код, который так работает.

Теперь мы запускаем цикл — для каждого человека в списке `people` ищем с помощью LINQ в списке адресов те записи, которые принадлежат этому человеку.

Каждый раз, когда мы станем выполнять `Where` на запросе `IEnumerable`, будет запускаться новая попытка получить все адреса из базы данных. Следующая строка:

```
var personAddress = address.Where(m => m.PersonID = a.PersonID);
```

станет причиной выполнения строки:

```
IEnumerable<PersonAddress> address =  
    db.Where(m => people.Contains(p => p.PersonID));
```

на каждом шаге цикла.

Потом эти полученные адреса будут фильтроваться в соответствии с нашим фильтром. А действительно ли нужно на каждом этапе цикла выбирать все данные из базы? Не думаю. Скорее всего, мы хотим получить адреса и людей из базы данных

один раз, а потом только в памяти фильтровать данные. Если вы не хотите, чтобы LINQ выполнял фильтрацию при каждом обращении, как в этом примере и как это было показано в листинге 10.1, то нужно конвертировать данные из `IEnumerable` в список:

```
List<Person> people = db.Where(/* фильтр */).ToList();
IEnumerable<PersonAddress> address =
    db.Where(m => people.Contains(p => p.PersonID)).ToList();
foreach (var p in people) {
    var personAddress = address.Where(m => m.PersonID == p.PersonID);
    // do something if address belongs to a person
}
```

Адреса здесь были сконвертированы в список, и эта конвертация выполнит фильтрацию, запросит данные из базы данных и сохранит их в памяти. Теперь все попытки вызывать `Where` и использовать переменную результата внутри цикла не будут приводить к тому, что системе придется фильтровать данные дважды.

Если учитывать производительность, то лучше привести адреса к `Lookup` с помощью метода `ToLookup`:

```
List<Person> people = SampleHelper.CreatePersons();
List<Address> addresses = SampleHelper.CreateAddresses();
var addr_lookup = addresses.ToLookup(m => m.PersonID);

foreach (var p in people)
{
    p.Addresses = addr_lookup[p.PersonID].ToList();
}
```

В этом случае `people` — это просто список, и мы его перебираем с помощью простого `foreach`. Когда нужно перебирать что-то в цикле, то для этого отлично подходит список `List`.

Внутри цикла мы должны искать адреса. Поиск адресов по списку приведет к тому, что придется перебирать все адреса в поиске нужных. `Lookup` работает как хеш-таблица, когда доступ к нужному элементу происходит быстро. Так что обращение к `Lookup` в цикле будет очень быстрым.

10.3. Доступ к данным

До сих пор мы получали доступ к данным просто перебором всех записей в результате. Но бывает необходимо найти первую запись в массиве, которая будет соответствовать условиям поиска. Например, давайте найдем первого человека с фамилией Иванов:

```
ivanov = people.Where(r => r.FirstName == "Иванов").First();
```

Если в массиве есть информация о человеке с фамилией Иванов, то будет возвращен первый соответствующий объект. Если же нет, то произойдут исключительные

ситуации, о которых мы говорили в *главе 8*. Чтобы исключительной ситуации не возникло, вместо метода `First` вызывайте метод `FirstOrDefault`. Он не будет генерировать ошибку, а вернет значение по умолчанию: для объектов — `null`, а для чисел — `0`:

```
ivanov = people.Where(r => r.FirstName == "Иванов").FirstOrDefault();
```

Когда нам нужна запись, мы можем вместо метода `Where` использовать метод `FirstOrDefault`:

```
ivanov = people.FirstOrDefault(r => r.FirstName == "Иванов");
```

Оба варианта будут возвращать один и тот же результат и работать идентично, поэтому вы можете выбрать тот, который вам больше нравится с точки зрения чтения кода.

Если нужно проверить массив на наличие в нем какого-либо значения, можно использовать метод `Exists` или воспользоваться тем же `FirstOrDefault`. Если `FirstOrDefault` возвращает `null`, то нужного нам объекта в массиве нет:

```
if (people.Where(r => r.FirstName == "Иванов").FirstOrDefault == null)
{
}
```

Можно также использовать метод `Any`, который возвращает `true`, если в списке есть хотя бы один элемент, который соответствует фильтру:

```
if (people.Any(r => r.FirstName == "Иванов"))
{
}
```

Если результатом может быть большое количество записей и вы хотите реализовать страничное отображение результата, то тут нам помогут два метода: `Skip(N)` и `Take(N)`. Первый из них говорит, что LINQ должен пропустить и не показывать первые *N* записей результата, а второй метод указывает, сколько нужно отобразить после этого.

Следующий пример пропускает первые 10 записей и отображает последующие 10, т. е. будут отображены элементы с 11-го по 20-й, если нумеровать с единицы:

```
IEnumerable<Person> result = people.Where(r => r.Age < 16).Skip(10).Take(10);
```

10.4. Использование LINQ для доступа к XML

Еще одна удобная и мощная возможность LINQ — это доступ к XML-файлам, которые получили большое распространение в качестве средства обмена данными. Для использования этих возможностей нужно подключить пространство имен `System.Xml.Linq`.

Для примера я создал небольшой XML-файл `person.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<export>
  <person FirstName="Иван" LastName="Иванов" Age="32">
    <address>
      <city>Ростов</city>
      <street>Королева</street>
    </address>
  </person>
  <person FirstName="Сергей" LastName="Петров" Age="50">
    <address>
      <city>Ростов</city>
      <street>Комарова</street>
    </address>
  </person>
  ...
  ...
</export>
```

ПРИМЕЧАНИЕ

Содержимое этого XML-файла, как и весь исходный код примера к этому разделу, можно найти в папке `Source\Chapter10\LinqXml\LinqXml` сопровождающего книгу электронного архива (см. приложение).

Допустим, нам нужно найти в нем всех людей до 16 лет. Мы уже знаем, что это можно делать с помощью метода `Where` любой коллекции, которая реализует интерфейс `IEnumerable`. То есть надо загрузить XML-файл в какой-то объект подходящего класса, а таким является `XElement` из пространства имен `System.Xml.Linq`.

У этого класса есть статичный метод `Load`, которому нужно передать имя файла, с которым вы хотите работать, а результатом будет объект типа `XElement`, через который как раз и можно будет работать с документом.

К элементам документа можно получить доступ через метод `Elements`, который в качестве параметра принимает имя интересующего нас тега. Первым в моем тестовом документе идет тег `person`, именно его и нужно передать. Метод `Elements` вернет коллекцию `IEnumerable`, с которой мы можем работать с помощью методов LINQ. Например:

```
XElement root = XElement.Load(@"d:\Temp\person.xml");
foreach (var item in root.Elements("person").Where(m =>
    Int32.Parse(m.Attribute("Age").Value) < 16)) {
    Console.WriteLine(item.Attribute("FirstName").Value);
}
```

Здесь внутри метода `Where` мы можем обратиться к значениям каждого атрибута:

```
m.Attribute("Age").Value
```

чтобы прочитать возраст — ведь возраст задан именно в атрибуте `Age`.

Усложним задачу: пусть мы хотим найти всех, кто живет в Ростове. Мы все так же должны перебирать людей, но в `Where` нужно смотреть на несколько уровней ниже этого тега. И такое возможно решить следующим образом:

```
XElement root = XElement.Load(@"d:\Temp\person.xml");
foreach (var item in root.Elements("person").Where(m =>
    m.Elements("address").Elements("city").First().Value == "Москва")) {
    Console.WriteLine(item.Attribute("FirstName").Value);
}
```

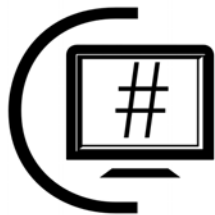
Здесь внутри метода `Where` я спускаюсь по дереву XML-документа с помощью методов:

```
Elements: m.Elements("address").Elements("city").First().Value.
```

XML и LINQ позволяют решить практически любую задачу, которая может возникнуть. Описать абсолютно все задачи в одной книге невозможно, я и пытаться не буду. Надеюсь, что этой информации вам для старта хватит.

XML — не самый быстрый формат файлов. У него очень много служебной информации: открытие тегов, закрытие тегов... Но этот формат достаточно часто используют, потому что он очень гибкий. С его помощью можно построить практически любую структуру и хранить любые данные.

ГЛАВА 11



Хранение информации

В этой главе мы рассмотрим хранение информации в различных местоположениях. Существует множество мест хранения данных, но нас будут интересовать два основных хранилища: реестр и файлы. И реестр, и файлы имеют свои преимущества и недостатки. Использование того или иного хранилища зависит от конкретной ситуации.

Мы также познакомимся здесь с различными типами данных и узнаем, как и где их хранить. Помимо этого, мы рассмотрим соответствующие классы и примеры их использования, максимально приближенные к практическим задачам, которые вам, может быть, придется решать в реальной работе.

11.1. Файловая система

Классы для работы с файловой системой находятся в пространстве имен `System.IO`, и для удобной работы с ними желательно подключить его к модулю. Основными из этих классов являются `Directory`, `File` и `Path`.

Давайте напишем небольшой пример файлового менеджера, который будет загружать список файлов из корня диска C: и отображать его в консоли.

Метод загрузки файлов `GetFiles()` показан в листинге 11.1.

Листинг 11.1. Метод получения списка каталогов и файлов

```
static void GetFiles(string path)
{
    foreach (string dir in Directory.GetDirectories(path))
    {
        if ((File.GetAttributes(dir) & FileAttributes.Hidden) ==
            FileAttributes.Hidden)
            continue;
    }
}
```

```
string dirname = System.IO.Path.GetFileName(dir);
Console.WriteLine(dirname.ToUpperInvariant());
}

foreach (string file in Directory.GetFiles(path))
{
    string filename = System.IO.Path.GetFileName(file);
    Console.WriteLine(filename.ToLowerInvariant());
}
}
```

Чтобы получить список каталогов по определенному пути файловой системы, используется статичный метод `GetDirectories()` класса `Directory`. Методу передается путь, по которому находится интересующее нас содержимое. В результате метод возвращает список строк с именами путей. Пути будут полными, поэтому, чтобы получить только имя, а не путь, можно использовать статичный метод `GetFileName()` класса `Path`.

Получение списка имен файлов происходит идентичным с именами каталогов способом, только тут используется статичный метод `GetFiles()`.

По умолчанию мы получаем полный список имен каталогов и файлов, включая скрытые и системные файлы. Чтобы в этом списке не отображались скрытые файлы, можно добавить следующую проверку:

```
if ((File.GetAttributes(s) & FileAttributes.Hidden) ==
    FileAttributes.Hidden)
    continue;
```

Метод `GetAttributes()` класса `File` получает атрибуты указанного в качестве параметра файла. С помощью операции `&` можно сложить результат с нужным атрибутом, и если результат равен атрибуту, то он установлен. Приведенный только что код проверяет, установлен ли атрибут невидимости `FileAttributes.Hidden`, и если да, то выполняется оператор `continue`. Если добавить этот код в цикле перед добавлением файла в список, то скрытые файлы будут пропускаться.

Чтобы проще было видеть разницу между каталогами и файлами, я при выводе имен каталогов отображаю их большими буквами: `dirname.ToUpperInvariant()`, а имена файлов — маленькими: `filename.ToLowerInvariant()`.

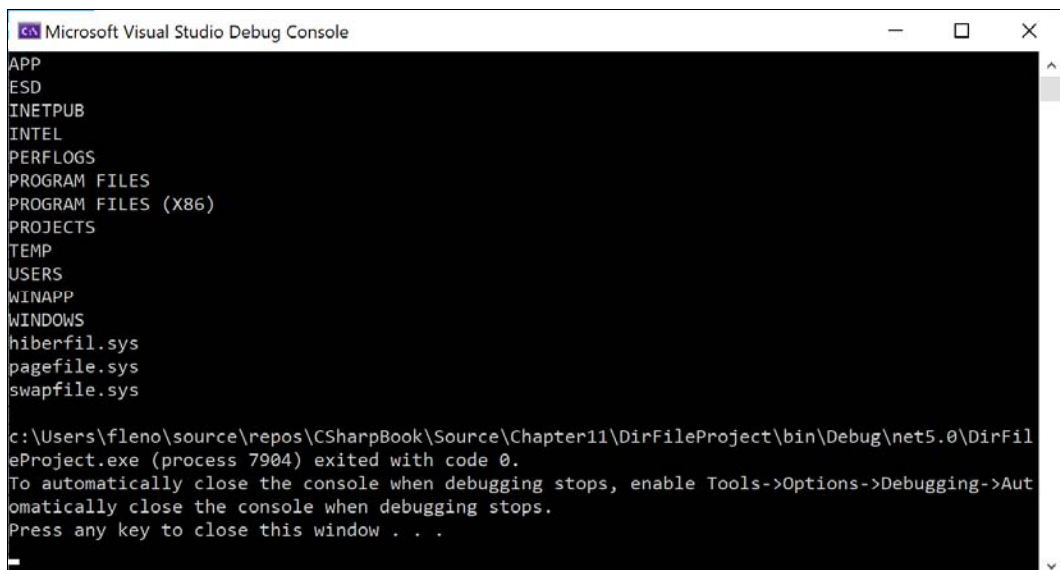
Результат работы программы показан на рис. 11.1.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter11\DirFileProject* сопровождающего книгу электронного архива (см. приложение).

Давайте рассмотрим, какие еще методы нам предоставляет класс `Directory` (они все статичные):

- ❑ `CreateDirectory()` — создает все вложенные каталоги в указанном в качестве параметра пути;



```
Microsoft Visual Studio Debug Console

APP
ESD
INETPUB
INTEL
PERFLOGS
PROGRAM FILES
PROGRAM FILES (X86)
PROJECTS
TEMP
USERS
WINAPP
WINDOWS
hiberfil.sys
pagefile.sys
swapfile.sys

c:\Users\fleno\source\repos\CSharpBook\Source\Chapter11\DirFileProject\bin\Debug\net5.0\DirFileProject.exe (process 7904) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Рис. 11.1. Результат работы программы

- ☐ `Delete()` — удаляет указанный каталог;
- ☐ `Exists()` — позволяет определить, существует ли указанный в качестве параметра каталог;
- ☐ `GetAccessControl()` — возвращает права доступа для указанного каталога;
- ☐ `GetCreationTime()` — возвращает время создания указанного каталога;
- ☐ `GetCurrentDirectory()` — возвращает текущий рабочий каталог приложения, куда будут сохраняться файлы, для которых не указан конкретный путь;
- ☐ `GetDirectoryRoot()` — возвращает информацию о томе;
- ☐ `GetFiles()` — возвращает имена файлов в указанном каталоге;
- ☐ `GetFileSystemEntries()` — возвращает все имена файлов и вложенных каталогов в указанном каталоге;
- ☐ `GetLastAccessTime()` — время последнего доступа к каталогу;
- ☐ `GetLastWriteTime()` — время последней записи в каталоге;
- ☐ `GetLogicalDrives()` — возвращает имена логических дисков в системе;
- ☐ `GetParent()` — возвращает родительский каталог;
- ☐ `Move()` — перемещает файл или каталог со всем содержимым в новое место;
- ☐ `SetAccessControl()` — назначает права доступа;
- ☐ `SetCreationTime()` — изменяет время создания;
- ☐ `SetCurrentDirectory()` — изменяет текущий каталог;
- ☐ `SetLastAccessTime()` — изменяет время последнего доступа;
- ☐ `SetLastWriteTime()` — изменяет время последней записи.

Теперь рассмотрим методы класса `File`, которые также являются статическими и доступны без создания объекта:

- ❑ `AppendAllText()` — добавляет указанный текст в файл. Если файл не существует, то он будет создан;
- ❑ `AppendText()` — добавляет текст в файл и возвращает объект `StreamWriter`, который можно использовать для дописывания в тот же файл дополнительной информации;
- ❑ `Copy()` — копирует указанный файл в новое положение;
- ❑ `Create()` — создает файл по указанному пути;
- ❑ `CreateText()` — создает файл для записи информации в формате UTF-8;
- ❑ `Decrypt()` — дешифрует содержимое файла;
- ❑ `Delete()` — удаляет указанный файл;
- ❑ `Encrypt()` — зашифровывает указанный файл;
- ❑ `Exists()` — проверяет существование указанного файла;
- ❑ `GetAccessControl()` — возвращает права доступа к файлу;
- ❑ `GetAttributes()` — возвращает атрибуты файла;
- ❑ `GetCreationTime()` — время создания указанного файла;
- ❑ `GetLastAccessTime()` — время последнего доступа к файлу;
- ❑ `GetLastWriteTime()` — время последней записи в файл;
- ❑ `Move()` — перемещает файл в новый каталог;
- ❑ `Open()` — открывает для чтения и изменения указанный файл;
- ❑ `OpenRead()` — открывает указанный файл для чтения;
- ❑ `OpenWrite()` — открывает указанный файл для записи;
- ❑ `OpenText()` — открывает файл для работы с ним как с текстом в формате UTF-8;
- ❑ `ReadAllLines()` — читает все строки файла;
- ❑ `SetAccessControl()` — назначает права доступа;
- ❑ `SetCreationTime()` — изменяет время создания;
- ❑ `SetLastAccessTime()` — изменяет время последнего доступа;
- ❑ `SetLastWriteTime()` — изменяет время последней записи;
- ❑ `WriteAllLines()` — создает файл и записывает в него указанный массив строк.

11.2. Текстовые файлы

Зная, как работать с файловой системой, и имея представление о функциях класса `File`, мы уже готовы написать еще один интересный пример. В нем мы решим простую задачу сохранения содержимого списка. Допустим, вам нужно сохранить

содержимое вводимых данных, и они настолько простые, что использовать базу данных смысла нет. Воспользуемся в таком случае для хранения данных файлом.

Итак, создаем простое консольное приложение. Полный исходный код его можно будет найти в папке Source\Chapter11\SaveStringArray сопровождающего книгу электронного архива (см. *приложение*), а здесь мы посмотрим только на самые интересные моменты.

В самом начале метода Main мы загружаем данные из файла в список:

```
string fullpath = Environment.GetCommandLineArgs()[0] + ".list";
List<string> items = new List<string>();

if (File.Exists(fullpath))
{
    items.AddRange(File.ReadAllLines(fullpath));
}
```

Сначала сохраняем в строковой переменной fullpath путь к текущему исполняемому файлу плюс расширение ".list". Переменная будет хранить имя файла, в котором находятся данные.

Прежде чем обращаться к файлу, желательно убедиться, что он существует, и в нашем примере мы делаем это с помощью статичного метода Exists() класса File. После этого можно загружать содержимое из файла.

Самый простой способ загрузить файл в виде строк в массив — воспользоваться статичным методом ReadAllLines() класса File. Он возвращает нам массив строк, который достаточно только добавить в список.

Теперь нашему примеру нужна возможность сохранения изменений, и для этого я создал отдельный метод SaveChanges, который вызывается в конце метода Main:

```
static void SaveChanges(string fullpath, List<string> items)
{
    StreamWriter sw = File.CreateText(fullpath);
    foreach (string s in items)
    {
        sw.WriteLine(s);
    }
    sw.Close();
}
```

В первой строке вызывается метод CreateText(), которому нужно передать имя создаваемого текстового файла, а в результате мы получаем объект класса StreamWriter, с помощью которого можно писать в этот файл. Для записи всех строк списка запускаем цикл foreach перебора содержимого списка, внутри которого с помощью метода WriteLine() объекта StreamWriter добавляем элемент в файл. После цикла желательно вызвать метод Close(), чтобы завершить работу с файлом и закрыть его.

Класс StreamWriter очень удобен, когда вам нужно работать с файлом как с текстом, т. е. когда файл содержит текст. Давайте посмотрим, какие еще методы есть у этого класса:

- ❑ `Flush()` — сбросить все изменения в файле на диск. Когда вы вызываете методы записи в файл, то в этот момент запись происходит в буфер, что позволяет повысить производительность. Если в этот момент выключить питание компьютера, то изменения могут не сохраниться. После вызова метода `Flush()` содержимое буферов физически записывается в файл;
- ❑ `Close()` — мы уже знаем, что этот метод закрывает файл. Он также сохраняет физически содержимое буферов, поэтому вам нет необходимости явно вызывать метод `Flush()` перед закрытием файла;
- ❑ `Write()` — сохраняет указанную в качестве параметра переменную в файл. Существует множество перегруженных вариантов этого метода для большинства основных типов данных;
- ❑ `WriteLine()` — сохраняет содержимое указанной в параметре переменной в файл и добавляет символы перевода каретки, т. е. информация будет сохранена в отдельной строке, а следующий вызов этого метода будет записывать данные в следующую строку.

ПРИМЕЧАНИЕ

Исходный код рассмотренного примера можно найти в папке *Source/Chapter11/SaveStringArray* сопровождающего книгу электронного архива (см. *приложение*).

Для чтения текстового файла можно использовать класс `StreamReader`. Этот класс похож на `StreamWriter`, но, судя по названию, предназначен для чтения текстовых данных из файла. Пример чтения данных из файла в список с использованием класса `StreamReader` приведен в листинге 11.2.

Листинг 11.2. Загрузка содержимого текстового файла

```
Console.WriteLine("Введите имя файла:");
string fileName = Console.ReadLine();

List<string> fileLines = new List<string>();
StreamReader reader = new StreamReader(fileName);
while (true)
{
    String s = reader.ReadLine();
    if (s == null)
        break;
    fileLines.Add(s);
}
reader.Close();
```

Мы работаем с консольными приложениями, поэтому тут есть неудобство — я прошу пользователя ввести в консоли имя файла, и ему нужно будет ввести полный путь. Если при вводе он допустит неточность, то возникнут проблемы, но ради простоты кода я не добавлял в него никаких проверок.

Теперь запускаем бесконечный цикл, внутри которого пытаемся прочитать очередную строку из файла с помощью метода `ReadLine()`. Если в файле есть строка, то она будет возвращена в качестве результата. Если больше ничего нет, то результатом будет `null`. Если достигнут этот `null`, то прерываем работу цикла.

ПРИМЕЧАНИЕ

Исходный код рассмотренного примера можно найти в папке `Source\Chapter11\TextFileReader` сопровождающего книгу электронного архива (см. приложение).

11.3. Бинарные файлы

Далеко не все файлы являются текстовыми и хранят текст. Попробуйте открыть с помощью Блокнота файл картинки, и вы увидите полный бред, потому что в файле, помимо читаемых символов, находится множество нечитаемых кодов, которые нельзя воспринимать как текст. Помимо этого, данные никак не сгруппированы, и там может не быть каких бы то ни было разделителей — типа, например, разделителей строк в текстовых файлах.

Для работы с файлом в бинарном виде используется класс `FileStream`. Давайте кратко рассмотрим свойства этого класса:

- ☐ `CanRead` — определяет, можно ли читать из файла;
- ☐ `CanSeek` — можно ли перемещаться по файлу;
- ☐ `CanWrite` — можно ли писать в файл;
- ☐ `Handle` — описатель операционной системы, связанный с файлом;
- ☐ `Length` — длина файла;
- ☐ `Position` — текущая позиция курсора.

Теперь посмотрим на методы класса:

- ☐ `Close()` — закрыть открытый файл;
- ☐ `Flush()` — сбросить все изменения из буфера на диск;
- ☐ `Read()` — прочитать блок данных;
- ☐ `ReadByte()` — прочитать один байт;
- ☐ `Seek()` — переместить курсор;
- ☐ `SetLength()` — установить размер файла;
- ☐ `Write()` — записать в файл блок данных;
- ☐ `WriteByte()` — записать в файл один байт.

Когда вы открываете файл, то в нем создается виртуальный курсор, который указывает на начало (на нулевой байт). В процессе чтения или записи данных этот курсор перемещается по файлу. Например, для чтения 10-го байта вы переместитесь внутри файла на 9 байтов, и после его прочтения курсор будет указывать уже на 11-й байт. При следующем вызове метода чтения или записи операция с данны-

ми будет происходить, начиная с этой позиции. Позицию можно изменить в любой момент с помощью метода `Seek()`.

Давайте напишем пример загрузки текстовых файлов с помощью объекта класса `FileStream`. Нужно учитывать, что этот объект загружает данные именно в бинарном виде, и мы не сможем загрузить информацию построчно. Вместо этого мы будем видеть весь файл как одну сплошную и плоскую поверхность. Как же тогда отобразить текстовый файл? Можно во время чтения файла искать внутри файла символы конца строки и перевода каретки — это байты с кодами 13 и 10. Но это не нужно, если воспользоваться компонентом `RichTextBox`. У этого компонента есть метод `AppendText()`, которому нужно передать строку, добавляемую в редактор текста. Если внутри добавляемой строки есть символы конца строки, то компонент автоматически разобьет все по отдельным строкам.

Итак, для примера на форме нам понадобится компонент `RichTextBox` и кнопка (или пункт меню), по нажатию на которую будет происходить загрузка файла. Код загрузки показан в листинге 11.3.

Листинг 11.3. Загрузка файла с помощью класса `FileStream`

```
// Отобразить окно выбора файла
OpenFileDialog ofd = new OpenFileDialog();
if (ofd.ShowDialog() != DialogResult.OK)
    return;

// вспомогательные переменные
byte[] buffer = new byte[100];
ASCIIEncoding ascii = new ASCIIEncoding();

// загрузка файла
FileStream fs = new FileStream(ofd.FileName,
    FileMode.Open, FileAccess.ReadWrite);
int readed = fs.Read(buffer, 0, 100);
while (readed > 0)
{
    richTextBox1.AppendText(ascii.GetString(buffer));
    readed = fs.Read(buffer, 0, 100);
}
```

Чтобы узнать имя загружаемого файла, я снова использую класс `OpenFileDialog` для отображения стандартного окна выбора файла. И после этого завожу две вспомогательные переменные:

- первая вспомогательная переменная — это массив из 100 значений типа `byte`. Этот массив необходим для хранения считываемой информации, потому что при чтении файла в бинарном виде метод возвращает нам данные в виде массива байтов, а не строк. Длина массива 100 не является хорошим решением, потому

что при загрузке большого файла будет происходить слишком много вызовов метода чтения, и каждый из них будет читать слишком маленькое количество информации. Слишком большое значение тоже станет не очень хорошим решением. Я не могу утверждать, какое значение наиболее оптимально, но на практике чаще выбирают 8 Кбайт, или 8192 байта;

- следующая вспомогательная переменная: `ascii` — имеет тип `ASCIIEncoding`. Это класс, который позволяет перекодировать ASCII-информацию в Unicode-строки. Платформа .NET оперирует Unicode-строками, а загружая информацию в бинарном виде, мы получаем байтовый массив, что соответствует ASCII-кодировке. С помощью объекта класса `ASCIIEncoding` мы будем переводить ASCII в Unicode, которую .NET понимает и любит.

Для загрузки файла создаем экземпляр класса `FileStream`. У этого класса есть множество перегруженных конструкторов. Я выбрал наиболее универсальный, принимающий три параметра:

- путь к файлу, который нужно загрузить;
- режим открытия файла. Этот параметр имеет тип перечисления `FileMode`, а значит, может принимать одно из следующих значений перечисления:
 - `CreateNew` — создать новый файл. Если он уже существует, произойдет исключительная ситуация;
 - `Create` — создать новый файл. Если файл уже существует, он будет перезаписан;
 - `Open` — открыть существующий файл. Если файл не существует, сгенерируется исключение;
 - `OpenOrCreate` — открыть существующий файл. Если он не существует, то будет создан;
 - `Truncate` — открыть существующий файл и обрезать его размер до нулевого;
 - `Append` — открыть существующий файл и переместить курсор записи в конец файла. Запись при этом будет происходить только в конец файла. Попытка переместить курсор на более раннюю позицию приведет к исключительной ситуации;
- режим доступа к файлу. Этот параметр имеет тип перечисления `FileAccess` и может принимать одно из следующих значений перечисления:
 - `Read` — разрешено чтение из файла;
 - `Write` — разрешена запись в файл;
 - `ReadWrite` — разрешены и чтение, и запись в файл.

В нашем примере открывается уже существующий файл (второй параметр равен `FileMode.Open`) для чтения и записи (третий параметр равен `FileAccess.ReadWrite`). В третьем параметре можно было бы указать и значение только для чтения, потому что писать в файл мы все равно не будем.

Теперь начинаем с контрольной попытки прочитать файл и вызываем метод `Read()` объекта `FileStream`. Этот метод получает три параметра:

- ❑ буфер, в который нужно записать прочитанные данные. Буфер должен быть в виде массива `byte`;
- ❑ смещение внутри буфера, начиная с которого будут сохраняться данные. Мы будем сохранять данные в буфер с самого начала, поэтому здесь указываем `0`;
- ❑ количество считываемых байтов. Мы передаем число `100`, чтобы полностью заполнить буфер.

Метод возвращает количество реально прочитанных байтов. Если достигнут конец файла, то результатом работы метода будет ноль. Чтобы прочитать весь файл, я запускаю цикл, который выполняется, пока количество прочитанных байтов больше нуля. Внутри цикла прочитанные данные преобразовываются из ASCII в Unicode-строку с помощью вызова `ascii.GetString(buffer)`, а результат преобразования добавляется в текстовое поле `RichTextBox`.

Напоследок нужно рассмотреть поближе еще один очень важный метод: `Seek()`. Метод получает два параметра: количество байтов, на которые нужно переместить курсор внутри файла, и переменную типа `SeekOrigin`, определяющую, откуда нужно начинать отсчет. Перечисление `SeekOrigin` имеет три значения:

- ❑ `Begin` — двигаться от начала файла;
- ❑ `Current` — двигаться от текущей позиции курсора;
- ❑ `End` — двигаться от конца файла.

Рассмотрим несколько примеров, чтобы лучше понять работу метода. Следующий вызов переместит курсор на отметку `10` байтов от начала файла:

```
fs.Seek(10, SeekOrigin.Begin);
```

Если дальше читать, начиная с этой позиции, то информация из первых `10` байтов не будет прочитана.

Если вам нужно переместиться в конец файла, чтобы начать добавление информации в файл, то курсор можно переместить следующим образом:

```
fs.Seek(0, SeekOrigin.End);
```

Следующий вызов перемещает курсор на `10` байтов назад относительно текущей позиции:

```
fs.Seek(-10, SeekOrigin.Current);
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source/Chapter11/FileStreamProject* сопровождающего книгу электронного архива (см. приложение).

11.4. XML-файлы

В программировании весьма популярным является формат хранения данных XML. Это связано с тем, что такой формат достаточно универсален и его легко переносить на другие платформы. Кроме того, открытый формат — уже не просто мода, но и конкурентное преимущество, а что может быть более открытым, чем текст, который пользователь может изменять даже без специализированного редактора.

Формат файлов XML — не новинка и существует уже очень давно. К его преимуществам относятся не только открытость, но и структурированность информации. Именно поэтому он постепенно набирает популярность и все глубже и глубже проникает в нашу жизнь. Только не стоит из-за популярности этого формата файлов использовать его везде и внедрять во все возможные области. Задействуйте XML там, где он реально может принести вам пользу.

Здесь надо отметить, что в последнее время все большую популярность приобретает формат JSON, но о нем мы поговорим отдельно чуть позже.

Благодаря популярности XML, его поддержка реализована во всех современных библиотеках и языках программирования, и .NET в этом плане не является исключением.

Итак, где может быть полезен формат XML? Он очень удобен для хранения конфигурации программ, которая должна переноситься с компьютера на компьютер. В UNIX-системах для хранения конфигурации чаще используют обычные текстовые файлы, которые обладают одним, но очень важным недостатком, — данные в них не имеют жесткой структуры.

11.4.1. Создание XML-документов

Для создания структуры XML в .NET служит класс `XmlTextWriter`. Он создает структуру, но для записи в файл должен использоваться другой класс — `FileStream`, с которым мы уже работали. Давайте рассмотрим класс `XmlTextWriter` на реальном примере и увидим его свойства и методы.

Итак, добавьте в меню формы пункты **Открыть**, **Сохранить** и **Сохранить как**. Код всех этих обработчиков я здесь приводить не стану, потому что его можно найти в папке `Source\Chapter11\XMLProject` сопровождающего книгу электронного архива (см. *приложение*), — вместо этого мы рассмотрим два интересных метода, которые используются для реализации обоих обработчиков: `SaveProject()` и `OpenProject()`. Первый из этих методов будет сохранять проект в файл. Его код можно увидеть в листинге 11.4.

Листинг 11.4. Метод сохранения XML-документа

```
void SaveProject()
{
    // создание потока записи и объекта создания XML-документа
    FileStream fs = new FileStream(filename, FileMode.Create);
    XmlTextWriter xmlOut = new XmlTextWriter(fs, Encoding.Unicode);
```

```
// старт начала документа
xmlOut.WriteStartDocument();
xmlOut.WriteComment("Этот файл создан для примера");
xmlOut.WriteComment("Автор: Михаил Фленов (www.flenov.info)");

// создаем корневой элемент
xmlOut.WriteStartElement("RosesPlant");
xmlOut.WriteAttributeString("Version", "1");

// цикл перебора всех роз и сохранения их
foreach (Rose item in roses)
    item.SaveToFile(xmlOut);

// закрываем корневой тег и документ
xmlOut.WriteEndElement();
xmlOut.WriteEndDocument();

// закрываем объекты записи
xmlOut.Close();
fs.Close();
}
```

Для компиляции примера нужно подключить два пространства имен:

```
using System.IO;
using System.Xml;
```

Первое из них нам уже знакомо — в нем реализованы функции работы с вводом/выводом, и оно нужно, чтобы сохранить XML-код в файл. Второе пространство имен необходимо непосредственно для создания структуры XML.

Теперь рассмотрим собственно метод `SaveProject()`. В самом начале мы создаем экземпляр класса `FileStream`, который используется здесь в качестве посредника, — именно через него будет сохраняться XML-структура документа. В качестве параметров методу передаем имя файла и `FileMode.Create`, чтобы файл создавался, а если он уже существует, то обнулялся.

Теперь создаем экземпляр класса `XmlTextWriter`. Его конструктору в качестве первого параметра указываем поток, через который будет происходить запись в файл, а в качестве второго параметра — кодировку файла. Я предпочитаю и вам советую использовать `Unicode`. Это необходимо не только для того, чтобы в файл можно было сохранить различные национальные символы, но и чтобы файл мог быть перенесен на другие платформы. Кодировка `Unicode` поддерживается не только на компьютерах `Windows`, но и в `Linux`, `Mac` и в других системах.

Существуют еще два перегруженных конструктора `XmlTextWriter`. Первый из них получает только один параметр — объект текстового файла класса `TextWriter`, а другой — имя файла и кодировку. Во втором варианте вам не нужно отдельно создавать поток, потому что `XmlTextWriter` возьмет функции записи на себя. Я предпо-

читаю указывать класс потока явно. Почему? Дело в том, что потоком может быть не только файл, но и просто оперативная память. Вы можете создать поток в памяти с помощью класса `MemoryStream`, который будет представлять оперативную память. Он схож по функциональности с `FileStream`, но, работая с ним, вы сохраняете данные не в файле, а в памяти, и можете потом этот объект без сохранения на диск отправить по сети или использовать другим способом.

В общем-то, уже можно приступить к записи в файл, но я бы порекомендовал изменить свойство `Formatting` класса `XmlTextWriter` на `Indented`. Это свойство отвечает за форматирование XML-тегов в файле. По умолчанию не будет никакого форматирования, что не очень удобно при редактировании файла напрямую. Лучше использовать форматирование `Indented`. В этом случае при сохранении дочернего раздела в XML-структуре при записи его тегов слева будут записываться пробелы. Но пробелы — это только символы по умолчанию, а вы можете установить для форматирования символ табуляции. За то, какой будет использоваться символ, отвечает содержимое свойства `IndentChar`.

Прежде чем начать записывать в файл структуру документа, нужно вызвать метод `WriteStartDocument()`. В этот момент в файл записывается заголовок документа, включающий тег, в котором находится информация о его версии и кодировке.

Теперь можно сохранить в файл комментарии. Это не является обязательным, но здесь я сохраняю комментарий, чтобы показать, как это делается, — с помощью метода `WriteComment()`, которому нужно передать в качестве единственного параметра строку комментария.

Теперь посмотрим, как происходит запись XML-тега. Все начинается с вызова метода `WriteStartElement()`. Метод начинает запись тега с именем, указанным в качестве параметра.

После этого вы можете сохранять атрибуты тега. Для записи атрибута нужно вызвать метод `WriteAttributeString()`, который получает два строковых параметра: имя атрибута и значение.

Чтобы завершить запись XML-тега, нужно вызвать метод `WriteEndElement()`. Этот метод не получает никаких параметров, а только добавляет в структуру XML-файла закрывающий тег для элемента.

Давайте рассмотрим вызов следующих трех строк:

```
xmlOut.WriteStartElement("RosesPlant");  
xmlOut.WriteAttributeString("Version", "1");  
xmlOut.WriteEndElement();
```

После вызова первой строки в XML-документе будет создан тег с именем `RosesPlant`:

```
<RosesPlant>
```

После вызова второй строки кода к этому тегу будет дописан атрибут с именем `Version` и значением `1`:

```
<RosesPlant Version="1">
```

После вызова последней строки в структуру XML-документа будет добавлено завершение последнего открытого тега. В нашем случае внутри тега мы не создали ничего, поэтому в файл будет сохранено сокращенное завершение тега:

```
<RosesPlant Version="1" />
```

Но внутри одного тега могут быть и другие теги, как в нашем примере. У нас после открытия тега `RosesPlant` нет вызова закрытия тега. Вместо этого запускается цикл, который перебирает все розы и вызывает методы сохранения их в XML-документ:

```
foreach (Rose item in roses)
    item.SaveToFile(xmlOut);
```

Если вы помните, то в нашем коде у розы не было никакого метода `SaveToFile()`. Не было, но мы его скоро напишем. Он будет сохранять в структуре XML-документа отдельно розы, и в результате документ станет выглядеть следующим образом:

```
<?xml version="1.0" encoding="utf-16"?>
<!--Этот файл создан для примера-->
<!--Автор: Михаил Фленов (www.flenov.info)-->
<RosesPlant Version="1">
    <Rose Name="Поза 0" X="106" Y="145" Width="50" Height="46" />
    <Rose Name="Поза 1" X="282" Y="58" Width="50" Height="46" />
</RosesPlant>
```

В нашем случае теги `Rose` внутри тега `RosesPlant` созданы как раз методом `SaveToFile()` класса `розы`.

Только после завершения цикла перебора всех роз мы вызываем метод завершения тега `RosesPlant` с помощью `WriteEndElement()` и тут же вызываем метод `WriteEndDocument()` для завершения создания документа.

Когда документ создан, нужно вызвать методы `Close()` для объектов класса `XmlTextWriter` и `FileStream`, при этом желательно сначала закрыть `XmlTextWriter`. Эти методы как раз и сбрасывают структуру документа непосредственно в файл.

Теперь посмотрим на метод `SaveToFile()`, который нужно написать у `розы`:

```
public void SaveToFile(XmlTextWriter xmlOut)
{
    xmlOut.WriteStartElement("Rose");
    xmlOut.WriteAttributeString("Name", Name);
    xmlOut.WriteAttributeString("X", X.ToString());
    xmlOut.WriteAttributeString("Y", Y.ToString());
    xmlOut.WriteAttributeString("Width", Width.ToString());
    xmlOut.WriteAttributeString("Height", Height.ToString());
    xmlOut.WriteEndElement();
}
```

Метод получает в качестве параметра `XmlTextWriter`, который содержит объект записи в файл. Объект `розы` добавляет новый тег, в атрибутах сохраняет свои свойст-

ва и закрывает тег. Все свойства приводятся к строке при сохранении в качестве атрибута.

Почему сохранение розы перенесено именно в класс розы? Почему нельзя было внедрить этот код в метод `SaveProject()`? Так можно было поступить, если бы метод `SaveProject()` использовал какое-то нестандартное сохранение. В нашем случае мы подразумеваем, что не только наша форма может захотеть сохранять данные в XML-файл, но и другие приложения и формы. Каждому приложению или форме достаточно вызвать метод сохранения розы, и не нужно постоянно повторять весь этот код.

С другой стороны, такой подход очень удобен с точки зрения расширяемости. Допустим, что вы добавляете новое свойство розы — например, цвет, и хотите, чтобы свойство сохранялось в файл. Достаточно изменить метод `SaveToFile()` тут же в классе розы и не надо будет искать внешний метод сохранения, вспоминать, где он находится и как используется.

Всегда старайтесь писать код сохранения в тех классах, свойства которых они сохраняют. Это не только красивее, но и удобнее.

11.4.2. Чтение XML-документов

В *разд. 10.4* мы уже читали данные из XML-документов с помощью LINQ. На этот раз рассмотрим немного другой способ.

Мы научились сохранять документ, и теперь пора узнать, как можно прочитать структуру XML-файла. Для чтения мы воспользуемся парой классов: `FileStream` и `XmlTextReader`. Первый из них загрузит содержимое файла, а второй — будет анализировать получаемый поток для предоставления нам в удобном виде тегов и атрибутов XML-документа. Класс `FileStream` нам уже знаком, давайте теперь на практике попробуем разобраться, как можно с помощью класса `XmlTextReader` прочитать XML-документ, а заодно познакомимся и с самим классом. Код метода чтения можно увидеть в листинге 11.5.

Листинг 11.5. Метод чтения XML-документа

```
void OpenProject(string newFilename)
{
    // инициализация классов для чтения
    FileStream fs = new FileStream(newFilename, FileMode.Open);
    XmlTextReader xmlIn = new XmlTextReader(fs);
    xmlIn.WhitespaceHandling = WhitespaceHandling.None;

    // переместимся в начало документа
    xmlIn.MoveToContent();

    // проверяем первый тег документа
    if (xmlIn.Name != "RosesPlant")
        throw new ArgumentException("Incorrect file format.");
    string version = xmlIn.GetAttribute(0);
```

```
// цикл чтения тегов документа
do
{
    // удалось ли прочитать очередной тег?
    if (!xmlIn.Read())
        throw new ArgumentException("Ошибка");

    // проверяем тип текущего тега
    if ((xmlIn.NodeType == XmlNodeType.EndElement) &&
        (xmlIn.Name == "RosesPlant"))
        break;

    // если это конечный элемент, то незачем проверять
    if (xmlIn.NodeType == XmlNodeType.EndElement)
        continue;

    // если это роза, то нужно читать ее параметры
    if (xmlIn.Name == "Rose")
    {
        Rose newItem = new Rose("", 0, 0);
        roses.Add(newItem);
        newItem.LoadFromFile(xmlIn);
    }
} while (!xmlIn.EOF);

// закрываем классы
xmlIn.Close();
fs.Close();

filename = newFilename;
designerPanel.Invalidate();
}
```

Для чтения данных мы здесь создаем экземпляр класса `FileStream`, указывая ему имя загружаемого файла и опцию `FileMode.Open`. После этого создаем экземпляр класса чтения XML-документа `XmlTextReader`, которому нужно передать объект потока, из которого объект класса `XmlTextReader` будет читать данные и анализировать теги.

После инициализации `XmlTextReader` мы, прежде чем начать чтение документа, изменяем только одно его свойство: `WhitespaceHandling`. Это свойство определяет, как нужно обрабатывать пробелы, имеющиеся в документе. Мы на пробелы обращать внимания не станем, потому что в них для нас нет значащей информации, поэтому отключим их обработку, установив свойство в `WhitespaceHandling.None`.

Теперь вызываем метод `MoveToContent()`, который заставляет анализатор перейти к первому значащему тегу. В этот момент анализатор `XmlTextReader` найдет в потоке первый тег, пропустив при этом заголовок документа и все комментарии, которые

были написаны в самом начале. В комментариях была записана только общая информация, они не содержали значащих данных, поэтому мы их и пропускаем.

Имя текущего тега можно прочитать в свойстве `Name`. Перейдя на начало данных, мы должны были попасть на тег с именем `RosesPlant`. Помните, что именно такой тег мы записывали первым в файл? Чтобы убедиться, что пользователь выбрал корректный XML-файл, созданный нашей программой или содержащий корректные данные, я проверяю, чтобы текущий тег был именно с таким именем. Так как с помощью метода `MoveToContent()` мы перешли на первый значащий тег, то эта проверка для корректного файла должна завершиться успехом. Если нет, то генерируется исключительная ситуация `ArgumentException`.

Когда мы сохраняли первый тег, то присвоили ему еще и атрибут со значением версии файла. Чтобы получить атрибут текущего тега, служит метод `GetAttribute()`. Этому методу передается индекс нужного атрибута или имя. Индексы атрибутов нумеруются с нуля, а версия была первым и единственным атрибутом, поэтому для его получения в качестве параметра указан ноль.

Теперь запускаем цикл, перебирающий все остальные теги. Причем нужно учитывать, что класс `XmlTextReader` будет возвращать нам не только начала тегов, но и их завершения, если они указаны в файле явно, а они нам не нужны. То есть при встрече открывающего тега мы должны быть готовы, что сейчас начнутся его данные, а завершающий тег мы будем просто игнорировать. Итак, внутри цикла сначала вызывается метод `Read()`. Метод переходит к следующему элементу дерева XML-документа.

Получив очередной элемент XML-дерева, мы делаем две проверки. Если текущий элемент является завершающим и имя тега `RosesPlant`, то мы точно добрались до конца файла. Тег с именем `RosesPlant` является корневым в нашем документе, и если мы дошли до конца этого тега, значит, достигнут конец файла. Если текущий элемент является завершающим тегом, то его свойство `EndElement` будет равно `true`. После этого проверяем, не является ли текущий элемент любым другим конечным элементом, и если это так, то тут загружать тоже ничего не нужно, и мы переходим к следующему шагу цикла.

Все предварительные проверки сделаны, поэтому остается только проверить, является ли текущий тег розой, т. е. равно ли его имя `"Rose"`. Если да, то это тег розы. В этом случае создаем новый экземпляр розы, добавляем его в список, и тут же ставляем эту розу загрузить свои свойства из XML-файла с помощью метода `LoadFromFile()`. Этот метод у класса еще не существует, но мы его создадим. Как и запись, чтение свойств будет происходить внутри класса розы. Таким образом, изменив содержимое класса, легко изменить и метод сохранения, потому что он находится в том же файле.

Цикл выполняется, пока свойство `EOF` не равно `true`. Это свойство станет равным `true`, когда наше чтение дойдет до конца файла. Когда файл прочитан, вызываем метод `Close()`, чтобы его закрыть.

Теперь наступила пора посмотреть на метод `LoadFromFile()`, который находится в классе розы и загружает данные объекта из XML-документа:

```
public void LoadFromFile(XmlTextReader xmlIn)
{
    try
    {
        Name = xmlIn.GetAttribute("Name");
        X = Convert.ToInt32(xmlIn.GetAttribute("X"));
        Y = Convert.ToInt32(xmlIn.GetAttribute("Y"));
        Width = Convert.ToInt32(xmlIn.GetAttribute("Width"));
        Height = Convert.ToInt32(xmlIn.GetAttribute("Height"));
    }
    catch (Exception)
    { }
}
```

Что тут добавить? Здесь последовательно читаются все атрибуты с помощью метода `GetAttribute()`. Этому методу мы передаем имя свойства, которое нас интересует. При этом все возможные ошибки просто гасятся, а ошибки тут могут быть разные. Во-первых, если атрибута с указанным именем нет, то произойдет исключительная ситуация. Так как все атрибуты являются строковыми, то при преобразовании их в числа опять может произойти ошибка, если XML-документ редактировался пользователем вручную.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter11\XmlProject` сопровождающего книгу электронного архива (см. приложение).

11.5. Поток *Stream*

Мы уже познакомились с классом `FileStream`, который позволяет работать с файлом в виде потока `Stream`. Тут есть небольшая путаница в терминах, потому что поток `Stream` не имеет ничего общего с потоками `Thread`, которые используются при многопоточности (см. главу 12). Проблема в том, что оба эти англоязычные термины: `Stream` и `Thread` — переводятся на русский язык как «поток».

Если говорить о потоке, который `Stream`, то он представляет собой лишь какой-то кусок данных, например:

- ❑ `FileStream` (файловый поток) — представляет собой просто файл и позволяет выполнять над ним операции ввода/вывода;
- ❑ `MemoryStream` (поток памяти) — реализует блок памяти. Такой поток можно тоже представить себе в виде файла, но хранящегося в памяти;
- ❑ `BufferedStream` (буферизированный поток) — обеспечивает дополнительную буферизацию при использовании операций чтения или записи над другими потоками.

Все эти три потока происходят от одного базового класса `Stream`. Именно этот класс определяет такие свойства, как:

- ☐ `CanRead` — можно ли читать данные;
- ☐ `CanSeek` — поддерживается ли метод `Seek()`;
- ☐ `CanTimeout` — поддерживается ли время ожидания;
- ☐ `CanWrite` — разрешена ли запись в поток;
- ☐ `Length` — размер потока данных;
- ☐ `Position` — позиция в потоке;
- ☐ `ReadTimeout` — время ожидания при попытке чтения, в миллисекундах;
- ☐ `WriteTimeout` — время ожидания при попытке записи, в миллисекундах.

Класс также определяет следующие методы, которые будут наследоваться потомками:

- ☐ `BeginRead()` — начать асинхронную операцию чтения;
- ☐ `BeginWrite()` — начать асинхронную операцию записи;
- ☐ `Close()` — закрыть поток;
- ☐ `Dispose()` — освободить все ресурсы, занимаемые потоком;
- ☐ `EndRead()` — закончить чтение;
- ☐ `EndWrite()` — закончить запись;
- ☐ `Flush()` — сбросить изменения на диск;
- ☐ `Read()` — прочитать порцию данных;
- ☐ `ReadByte()` — прочитать только один байт;
- ☐ `Seek()` — перемещение курсора по потоку;
- ☐ `SetLength()` — установить новый размер потока;
- ☐ `Synchronized()` — статический метод, создающий защищенный для многопоточного программирования объект потока `Stream`;
- ☐ `Write()` — записать блок данных;
- ☐ `WriteByte()` — записать один байт.

С большинством этих свойств мы уже знакомились на практике, когда в начале главы рассматривали работу с файлами (файловые потоки). Я специально оставил рассмотрение базового потока на конец главы, потому что для него нельзя написать примера. Класс `Stream` является абстрактным, и поэтому вы не можете создать объект этого класса. Вместо этого нужно использовать потомки.

11.6. Потоки *MemoryStream*

Класс `MemoryStream` предоставляет нам блок в памяти, к которому можно осуществить доступ с помощью функций чтения и записи, как мы это делали с файлами. И это не удивительно, ведь `MemoryStream` является потомком от класса `Stream`.

Давайте напишем пример, в котором сохраним в памяти строку (листинг 11.6).

Листинг 11.6. Работа с потоком памяти

```
const string STRING_EXAMPLE = "Эту строку поместим в память";

// превращаем строку в массив символов
UnicodeEncoding unicode = new UnicodeEncoding();
byte[] str = unicode.GetBytes(STRING_EXAMPLE);
int string_size = unicode.GetByteCount(STRING_EXAMPLE);

// создаем поток MemoryStream и записываем в него данные
MemoryStream ms = new MemoryStream(string_size);
ms.Write(str, 0, string_size);
// перемещаемся в начало потока
ms.Seek(0, SeekOrigin.Begin);

// создаем буфер
byte[] buffer = new byte[string_size];

// читаем поток
ms.Read(buffer, 0, string_size);
Text = unicode.GetString(buffer);
```

Чтобы было удобнее работать со строкой, я помещаю ее в константу. Сразу же за этим начинается самое интересное. Строки в .NET хранятся в кодировке Unicode, где каждый символ занимает два байта, а класс `MemoryStream` работает с массивами одиночных байтов. Получается, что для записи строки в поток нам нужно сначала превратить ее в массив байтов. Для этого можно воспользоваться классом `UnicodeEncoding`. У него есть метод `GetBytes()`, который возвращает нужный нам массив байтов. Помимо этого, у него есть метод `GetByteCount()`, с помощью которого можно узнать размер строки в байтах, а не в символах. Именно это и делается в первом блоке кода.

После этого работа с потоком памяти превращается в дело техники. Создаем объект класса `MemoryStream` и сохраняем в него массив байтов. Теперь, чтобы прочитать данные из потока, нужно перейти в его начало. Как и в случае с файловыми потоками, во время записи по потоку перемещается курсор, указывающий на позицию, в которой будет происходить чтение и запись. Записав данные, курсор окажется в конце блока записи, и если попытаться прочитать что-то в этой позиции, то мы получим пустоту.

Чтобы перейти на начало блока, используем метод `Seek()` потока памяти. В качестве первого параметра указываем нулевое смещение, а в качестве второго — указываем, что нужно двигаться от начала файла: `SeekOrigin.Begin`. Если вы хотите прочитать данные, начиная с 10-го байта от начала потока, то нужно написать следующую строку кода:

```
ms.Seek(10, SeekOrigin.Begin);
```

Теперь создаем буфер для чтения и читаем данные с помощью метода `Read()`. Будьте внимательны, данные читаются в виде массива байтов, и для превращения их в .NET-строку (Unicode) можно использовать метод `GetString()` класса `UnicodeEncoding`. В нашем примере результат преобразования сохраняем в свойстве `Text` текущей формы, т. е. в заголовке текущего окна.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter11\MemoryStreamProject` сопровождающего книгу электронного архива (см. *приложение*).

11.7. Сериализация

Сериализация позволяет сохранить состояния объекта в *потоке* (том, который `Stream`). Сохранив состояние объекта, мы можем закрыть программу, а при повторном ее запуске восстановить прежнее состояние. Состояние объектов можно передавать и по сети, чтобы восстанавливать его на другом компьютере. Это может пригодиться при распределенных расчетах, когда объект хранит данные для расчета, а мы с помощью сериализации передаем их на другой компьютер.

Классическим хранилищем для свойств объекта является файл. Да, вы можете сохранить в файле все свойства самостоятельно и потом восстановить их с помощью классов, которые мы рассматривали в этой главе, но сериализация реализуется намного проще и может сэкономить вам драгоценное время, которого всегда не хватает.

Чтобы состояние объекта можно было сохранять в потоке `Stream` с помощью сериализации, перед объявлением класса необходимо поставить атрибут `[Serializable]`. Например, давайте объявим упрощенный вариант класса для хранения данных о человеке, состояние которого можно сохранять с помощью сериализации:

```
[Serializable]
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public int Age { get; set; }
    public DateTime Birthday { get; set; }
}
```

Конечно, хранить одновременно и возраст, и дату рождения не имеет смысла, потому что возраст всегда можно вычислить по дате рождения, но я добавил оба поля для разнородности данных. Самое главное в объявлении — это первая строка, в которой находится атрибут `[Serializable]`.

Итак, создадим новое консольное приложение, в котором объявим объект `Person`, заполним его свойства, сериализуем и сохраним в файл (листинг 11.7).

Листинг 11.7. Сохранение состояния объекта

```

Person person = new Person();

person.FirstName = "Михаил";
person.LastName = "Фленов";
person.Age = 44;
person.Birthday = new DateTime(1976, 8, 11);

StreamWriter sw = File.CreateText("person.txt");
sw.WriteLine(JsonSerializer.Serialize(person));
sw.Close();

```

Сначала здесь создается объект класса `Person`, и в него я сохраняю данные о себе. После этого я создаю текстовый файл, как мы это уже делали в *разд. 11.2*. И вот теперь начинается самое интересное — сериализация. Для этого я использовал статичный метод `Serialize` класса `JsonSerializer`. Метод возвращает строку, в которой объект будет представлен в формате JSON и которую можно сохранить в файл, — именно это я и делаю.

Для работы с `JsonSerializer` нужно подключить следующее пространство имен:

```
using System.Text.Json;
```

Теперь в папке `bin\Debug\net5.0\` должен появиться файл `person.txt`. Откройте его в любом текстовом редакторе и посмотрите на его содержимое:

```
{ "FirstName": "\u041C\u0438\u0445\u0430\u0438\u043B", "LastName": "\u0424\u043B\u0435\u043D\u043E\u0432", "Age": 44, "Birthday": "1976-08-11T00:00:00" }
```

Да, большая строка, которая, на первый взгляд, не очень легко читается просто потому, что в ней нет пробелов. Это формат по умолчанию — он достаточно компактный, чтобы при передаче по сети или при сохранении в файл сериализованная версия занимала как можно меньше места.

Средства управления сериализацией предоставляют нам возможность осуществлять сохранение в более читабельном формате. Для этого методу `Serialize` в качестве второго параметра нужно передать объект `JsonSerializerOptions` с параметрами для превращения объекта в строку. В следующем примере устанавливается параметр `WriteIndented`, что сделает сериализованную строку более приятной для чтения:

```

JsonSerializerOptions options = new JsonSerializerOptions();
options.WriteIndented = true;
sw.WriteLine(JsonSerializer.Serialize(person, options));

```

Запустите программу, она перезапишет файл новой версией, и если теперь взглянуть на его содержимое, то вы должны увидеть что-то типа:

```

{
  "FirstName": "\u041C\u0438\u0445\u0430\u0438\u043B",
  "LastName": "\u0424\u043B\u0435\u043D\u043E\u0432",

```

```
"Age": 44,  
"Birthday": "1976-08-11T00:00:00"  
}
```

Это уже намного проще читать. Фигурные скобки задают границы описания объекта. Внутри скобок явно идет список в формате:

"имя свойства": Значение

Элементы списка разделены запятыми. Если значение является числом, то оно указывается в явном виде. Если это строка, то она в двойных кавычках. Строковые значения хранятся в каком-то непонятном формате, но на самом деле это кодировка Unicode.

Теперь посмотрим на код десериализации, который показан в листинге 11.8.

Листинг 11.8. Загрузка состояния объекта

```
StreamReader sr = File.OpenText("person.dat");  
string str = sr.ReadToEnd();  
sr.Close();  
  
Person restoredPerson = JsonSerializer.Deserialize<Person>(str);  
  
Console.WriteLine("First name: " + restoredPerson.FirstName);  
Console.WriteLine("Last name: " + restoredPerson.LastName);  
Console.WriteLine("Age: " + restoredPerson.Age);  
Console.WriteLine("Birthday: " + restoredPerson.Birthday);
```

На этот раз, чтобы прочесть содержимое файла, мы создаем объект `StreamReader`, читаем все содержимое файла в строковую переменную с помощью метода `ReadToEnd` и закрываем файл.

Самое интересное в коде — это вызов метода `Deserialize()` класса `JsonSerializer`. Метод получает только один параметр — строку. Метод читает данные в поисках свойств сохраненного объекта, формирует этот объект и возвращает его в виде результата работы. Нам нужно только помочь методу и в угловых скобках указать тип класса, который должен получиться в результате, потому что в JSON-файле нет имени класса и десериалайзер не знает, какого класса объект получится.

Далее метод сам создает объект нужного класса, поэтому мы не инициализируем переменную `restoredPerson`, в которую сохраняется результат десериализации. Создается переменная именно класса `Person`, а нам остается только получить результат и сохранить его в `restoredPerson`.

Чтобы убедиться, что у прочитанного из файла объекта такие же свойства, как мы задали в коде перед сохранением, я вывожу значения всех полей в консоль.

Объекты можно сохранять не только в файлах, но и в потоке памяти `MemoryStream`. Например, перед выполнением каких-то расчетов мы можем сохранить состояние

объекта в потоке памяти и восстановить состояние после расчетов. В такие моменты потоки в памяти предоставляют нам всю свою мощь и скорость доступа.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter11\SerializeProject* сопровождающего книгу электронного архива (см. приложение).

11.7.1. Отключение сериализации

Далеко не все свойства должны сохранять свое состояние. Например, свойства, которые зависят от внешних факторов или рассчитываются во время выполнения программы, нельзя или не имеет смысла сохранять. Соответственно, в нашем классе *Person* я не зря ввел поле *Age*. Оно как раз хорошо показывает вариант, когда сериализация испортит работу приложения. Допустим, что вы сохранили состояние объекта в потоке и восстановили его через год — что произойдет в этом случае? У нашего человека обязательно пройдет день рождения, и возраст изменится, поэтому поле *Age* окажется неактуальным. Такое поле сохранять нельзя, оно должно рассчитываться автоматически.

Вы можете сказать, что в реальном приложении вы вообще не будете создавать поле возраста, потому что оно должно быть только для чтения, и его аксессор *get* должен вычислять результат, а аксессор *set* — отсутствовать.

Хорошо, давайте тогда рассмотрим класс любого сетевого сервера — например, FTP-сервера. Ему желательно знать, сколько клиентов сейчас подключено, чтобы контролировать нагрузку, да и просто для удобства. Количество клиентов хранится в отдельном классе статистики, и веб-сервер будет его изменять. Нужно ли сохранять это свойство? Опять же — нет, потому что оно зависит от внешних факторов. После восстановления состояния объекта такого количества соединений может и не быть, что приведет к выдаче некорректной информации.

Но продолжим рассматривать сериализацию на примере класса, описывающего данные человека:

```
[Serializable]
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int Age {
        get { return DateTime.Today.Year - Birthday.Year; }
    }
    public DateTime Birthday { get; set; }
}
```

Теперь *Age* — это поле, которое можно только читать, и возраст теперь рассчитывается. Так что если запустить приложение и посмотреть на результат, то в файле все равно будет находиться возраст *Age*.

Теоретически никакой проблемы здесь нет — ведь даже при восстановлении объекта при попытке чтения свойства `Age` будет возвращаться расчетное значение. Проблема возникнет в том случае, если вы передаете эти данные в другое приложение, где описан такой же класс, и у него программист по какой-то причине делает поле `Age` изменяемым.

JSON — это универсальный формат, и в мире веб-программирования достаточно часто можно увидеть случаи, когда код на C# создает на сервере объекты, передает их браузеру, и там они обрабатываются с помощью JS. Разные языки могут стать причиной разных реализаций.

Кроме того, это же лишняя информация, которая занимает дефицитное пространство на диске или будет передаваться по сети.

Чтобы .NET не сохранял сериализованную информацию в файл, нужно опцию `IgnoreReadOnlyProperties` установить в `true`, как в следующем примере:

```
JsonSerializerOptions options = new JsonSerializerOptions();
options.WriteIndented = true;
options.IgnoreReadOnlyProperties = true;
sw.WriteLine(JsonSerializer.Serialize(person, options));
```

А что если у класса есть открытая переменная? Допустим, что у нашего класса имеются поля:

```
[Serializable]
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public int Age {
        get { return DateTime.Today.Year - Birthday.Year; }
    }
    public DateTime Birthday { get; set; }

    public string City;
    public string Country;
}
```

Здесь `Country` и `City` просто являются открытыми (`public`) переменными, а не свойствами. По умолчанию они тоже не будут сериализоваться. У меня пока не было такого случая, чтобы нужно было использовать просто поля, — я предпочитаю и всегда использую именно свойства, но допустим, что кто-то сделал поля, и вы уже не можете ничего изменить. Чтобы все же сохранить их состояние — ведь это данные, и они открытые, — мы должны включить еще одну опцию: `IncludeFields`, как показано в следующем примере:

```
JsonSerializerOptions options = new JsonSerializerOptions();
options.WriteIndented = true;
options.IgnoreReadOnlyProperties = true;
options.IncludeFields = true;
sw.WriteLine(JsonSerializer.Serialize(person, options));
```

Отлично, теперь в файле будут не только свойства, но и город, и страна.

Тем не менее я все же рекомендую использовать свойства.

11.7.2. Сериализация в XML

До сих пор сериализация происходила в формате JSON, но с тем же успехом мы можем сохранять состояние объекта и в формате XML. Для этого сначала нужно подключить пространство имен:

```
using System.Xml.Serialization;
```

Сериализация в формате XML работает несколько иначе, чем в формате JSON, и это немного неудобно. Класс вроде бы выглядит так же, но работает по-другому.

Во-первых, класс `Person` должен быть открытым, поэтому убедитесь, что перед объявлением класса стоит слово `public`. Теперь сам код сериализации:

```
StreamWriter sw = File.CreateText("person.dat");
XmlSerializer serializerout = new XmlSerializer(typeof(Person),
    new Type[] { typeof(Person) });
serializerout.Serialize(sw, person);
sw.Close();
```

Здесь мы используем уже не статичный метод, а создаем сначала экземпляр класса, которому нужно передать тип сериализуемых данных, — в нашем случае: `typeof(Person)`. Вот теперь мы можем вызывать метод `Serialize`, и он получает два параметра: поток, куда нужно сохранить результат, и объект.

В результате в файле будет сохранен XML-вариант представления объекта:

```
<?xml version="1.0" encoding="utf-8"?>
<Person
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <City>Newmarket</City>
  <Country>Канада</Country>
  <FirstName>Михаил</FirstName>
  <LastName>Фленов</LastName>
  <Birthday>1976-08-11T00:00:00</Birthday>
</Person>
```

Хотя в этот раз у нас не было опций сериализации, но и свойство возраста не попало в результирующий файл. Вот такое тут отличие в работе кода.

Теперь посмотрим, как мы можем прочитать файл. Хотя я мог бы использовать уже существующий экземпляр класса `XmlSerializer`, я все же решил создать новый, чтобы подчеркнуть, что они не различаются:


```
XmlSerializer serializerin = new XmlSerializer(typeof(Person),  
    new Type[] { typeof(Person) });  
StreamReader fsin = File.OpenText("person.dat");  
Person restoredPerson = (Person)serializerin.Deserialize(fsin);  
fsin.Close();
```

После создания `XmlSerializer` создается поток и открывается файл. Теперь мы готовы вызвать метод десериализации `Deserialize`, которому передается поток и из которого нужно читать данные. В отличие от сериализации JSON, здесь мы не передаем тип данных, чтобы получить нужный объект, а получаем объект и приводим его к нужному.

11.7.3. Особенности сериализации

Сериализация может происходить целыми графами. С помощью графа вы можете представить взаимосвязь объектов, и все эти связи будут сохранены в файле. Например, у объекта `Car` может быть свойство двигателя `Engine`. Сериализация сохранит не только сам объект машины, но и его двигатель. Машина может происходить от объекта `Тарантайка`, ее свойства также будут сохранены. Наследственность тоже может быть представлена в виде графа, ведь это взаимодействие объектов.

Кстати, если вы не знаете, что здесь такое *граф* (и это не муж графини), — не страшно. Главное понимать, что сериализация сохранит все связанные объекты, если у этих объектов установлен атрибут `[Serializable]`.

Следующая особенность кроется в наследовании. Атрибуты сериализации не наследуются, поэтому если базовый класс объявлен с атрибутом `[Serializable]`, то дочерний не станет сериализуемым, пока у него тоже явно не будет указан атрибут `[Serializable]`. При наследовании будьте внимательны — если нужно сохранить возможность сериализации, то следует для наследника явно указать соответствующий атрибут.

Теперь поговорим о формате файлов сериализуемых данных. Бинарный формат более компактный и работает достаточно быстро, но если вы предпочитаете открытые стандарты и XML-формат, то минимальные изменения в коде и использование класса `XmlSerializer` позволят вам сохранять состояние объекта или графа объектов в XML-файл.

Какой же формат сериализации выбрать для своего приложения? Трудно давать какие-то рекомендации, поэтому я опишу пару преимуществ каждого метода, а вы уже выбирайте согласно требованиям задачи. Если вам не нужна открытость, то я бы выбрал бинарный формат, потому что он должен работать быстрее. В случае с XML загрузчику придется тратить дополнительное время на анализ структуры XML-тегов.

Если вы выберете открытый формат XML, то сможете написать собственный загрузчик или конвертер данных под другие приложения для использования с другими классами. С бинарным форматом, мне кажется, такое реализовать будет немного сложнее.

А что если мы хотим сохранить целый массив объектов? Метод сериализации получает только одиночный объект, и массив передать не получится. Проблема решается достаточно просто — нужно только при создании массива указать тип сериализуемых данных. Пример сохранения массива показан в листинге 11.9. Чтобы пример был интереснее, сериализация происходит в XML-файл.

Листинг 11.9. Пример сериализации массива

```
static void Main(string[] args)
{
    // сохранение данных
    List<Person> persons = new List<Person>();
    persons.Add(new Person("Иванов", "Иван"));
    persons.Add(new Person("Петров", "Петр"));

    // создание файла
    FileStream fsout = new FileStream("peoples.dat",
        FileMode.Create, FileAccess.Write);
    // сериализация данных
    XmlSerializer serializerout = new XmlSerializer(typeof(List<Person>),
        new Type[] { typeof(Person) });
    serializerout.Serialize(fsout, persons);
    fsout.Close();

    // загрузка данных
    List<Person> persons1 = new List<Person>();
    FileStream fsin = new FileStream("peoples.dat", FileMode.Open,
        FileAccess.Read);

    // десериализация данных
    XmlSerializer serializerin = new XmlSerializer(typeof(List<Person>),
        new Type[] { typeof(Person) });
    persons1 = (List<Person>)serializerin.Deserialize(fsin);
    fsin.Close();

    // проверяем
    foreach (Person p in persons1)
        Console.WriteLine(p.FirstName);
    Console.ReadLine();
}
```

Для краткости кода я использовал в этом примере консольное приложение. Класс `Person` должен быть объявлен как публичный и должен иметь конструктор по умолчанию (без параметров), иначе попытка создать такой класс сериализации, как `XmlSerializer`, завершится исключительной ситуацией.

Теперь посмотрим на самое интересное в этом примере — на инициализацию объекта класса `XmlSerializer`:

```
XmlSerializer serializerout =  
    new XmlSerializer(typeof(List<Person>),  
        new Type[] { typeof(Person) });
```

В качестве первого параметра передаем тип данных массива, а во втором параметре передаем массив типов данных, которые могут находиться в массиве и которые нужно сериализовать. Если в массиве находятся разные объекты, то вы можете указать те типы данных, которые должны сохраняться. Если нужно обрабатывать полностью массив, не обращая внимания на типы объектов, можно использовать конструктор, который получает только тип данных массива:

```
XmlSerializer serializerout =  
    new XmlSerializer(typeof(List<Person>));
```

Чтобы увидеть пример в действии, сначала создайте массив и сохраните его в файл. После этого объявите другой массив, в который загружаются данные из того же файла. Для чистоты эксперимента при загрузке используйте совершенно новые объекты, никак не связанные с теми, которые использовались при сохранении данных.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter11\SerializeArray* сопровождающего книгу электронного архива (см. приложение).

11.7.4. Управление сериализацией

Возможностей, предоставляемых методами сериализации платформы .NET, достаточно для решения большинства задач сохранения состояний объектов. Большинство, но не всех. Могут возникнуть ситуации, когда нужно будет получить контроль над сохраняемой информацией или ее представлением. До появления .NET 2.0 для управления сериализацией приходилось реализовывать интерфейс *ISerializable*. Интерфейс *ISerializable* мы рассматривать не станем, потому что я предпочитаю использовать современные техники и методы программирования и вам рекомендую. Чтобы не было соблазна использовать что-то устаревшее, об этом лучше и не знать. Лично я уже забыл о существовании интерфейса *ISerializable*.

В современных приложениях лучше пользоваться следующими атрибутами:

- ☐ `[OnSerializing]` — позволяет указать метод, который будет вызываться при сериализации объекта;
- ☐ `[OnSerialized]` — позволяет указать метод, который будет вызываться сразу после завершения сериализации объекта;
- ☐ `[OnDeserializing]` — позволяет указать метод, который будет вызываться при десериализации объекта;
- ☐ `[OnDeserialized]` — позволяет указать метод, который будет вызываться сразу после завершения десериализации объекта.

Если ваш класс объявлен с атрибутом `[Serializable]` и система может сохранять состояние объекта, то вы можете использовать упомянутые атрибуты для указания методов, которые будут выступать как обработчики событий при сериализации и десериализации.

Вы можете задействовать любое количество из этих атрибутов. В примере из листинга 11.10 показан класс `Person`, в который я добавил методы для всех упомянутых атрибутов, но используется только один метод: `OnDeserializedMethod`.

Листинг 11.10. Управление сериализацией через атрибуты

```
[Serializable]
public class Person
{
    ...
    ...
    bool DeserializedVersion = false;

    [OnSerializing]
    internal void OnSerializingMethod(StreamingContext context)
    {
    }

    [OnSerialized]
    internal void OnSerializedMethod(StreamingContext context)
    {
    }

    [OnDeserializing]
    internal void OnDeserializingMethod(StreamingContext context)
    {
    }

    [OnDeserialized]
    internal void OnDeserializedMethod(StreamingContext context)
    {
        DeserializedVersion = true;
    }
}
```

Метод `OnDeserializedMethod()` объявлен с атрибутом `OnDeserialized` и будет вызываться сразу после десериализации объекта. В нашем случае метод изменяет переменную `DeserializedVersion` на `true`. Так мы можем узнать, создан ли объект в программе или десериализован из потока.

Возможности управления сериализацией с помощью атрибутов получаются действительно безграничными (the sky is the limit).

Давайте теперь представим другую ситуацию с изменением объекта. Допустим, что мы выпустили на рынок продукт, который некоторое время успешно продается, и пользователи с ним работают — сохраняют у себя на диске какие-то файлы с сериализованными объектами. И тут мы выпускаем новую версию продукта, в которой у класса `Person` появляется новое поле. Что произойдет? Если мы просто объявим поле и не позаботимся об изменениях, то все файлы, сохраненные пользователями, станут бесполезными — новая версия не сможет восстановить состояние объектов по этим файлам.

Если вы объявляете новое поле, то самый простой способ предотвратить возможную ошибку — объявить это поле как опциональное. В этом случае если во время восстановления состояния объекта будет выяснено, что какое-то поле не существует в файле, то исключительная ситуация генерироваться не будет.

Чтобы объявить поле как опциональное, перед его объявлением нужно поставить атрибут `[OptionalField]`:

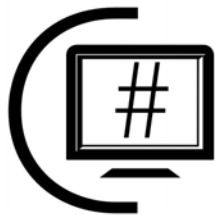
```
[Serializable]
public class Person
{
    // здесь идет объявление старых полей
    ...
    ...

    // новое опциональное поле
    [OptionalField]
    public string Address;
}
```

В этом примере мы добавили новое поле для хранения адреса проживания человека. В старой версии его не было, но благодаря атрибуту `[OptionalField]` старые файлы сериализации будут загружены без проблем. Ваша задача — только правильно обрабатывать такую ситуацию, когда восстановленный объект не обработал новое поле, т. е. не восстановил его состояние. Это состояние может быть задано по умолчанию или запрошено у пользователя.

Атрибут `[OptionalField]` является очень удобным средством для решения проблемы добавления поля, но оно не идеально. Если в ваше приложение добавлено сразу множество полей, то следует задуматься о создании новой версии файла сериализации. Впрочем, это уже совершенно другая история. То, как поддерживать версии форматов файлов, зависит от предпочтений программистов и нигде и никак не регламентируется. В каждом отдельном случае программисты поступают по-своему.

ГЛАВА 12



Многопоточность

Прошли те времена, когда ОС и приложения были однопоточными и могли одновременно выполнять только один процесс. Современные ОС поддерживают *многопоточность*, и не удивительно, что эта технология реализована в .NET. По умолчанию при запуске приложения создается процесс, который выполняет главный поток, в котором и начинается выполнение метода `Main()`. Но главный поток может создавать вторичные потоки, которые будут выполняться параллельно основному.

Поток — это путь выполнения кода. Главный поток приложения начинает свой путь с метода `Main()`, последовательно выполняя его команды. Создавая дочерние потоки, вы должны указать свою точку входа (свой метод), начиная с которой будет происходить выполнение вторичного потока.

Без вторичных потоков реализация некоторых задач может оказаться весьма сложным занятием. Допустим, приложение должно ожидать данные по сети. Если просто вызвать функцию ожидания данных в синхронном режиме из основного потока, то выполнение потока остановится, пока данные не поступят. А если данные не поступят вовсе? Приложение не будет отвечать на другие события, потому что оно занято ожиданием. У вас были случаи, когда вы что-нибудь делаете в программе, а она замирает и перестает отображать в окне информацию? Вполне возможно, что проблема была связана как раз с ожиданием какого-либо действия.

Проблема решается вызовом функции чтения сетевых данных в отдельном потоке или асинхронно, без блокирования основного потока, чтобы приложение могло продолжать заниматься своими делами.

Еще недавно многопоточность в компьютерах достигалась искусственным путем, потому что системы были однопроцессорными, а процессор не мог выполнять две и более задачи одновременно. Многопроцессорные системы существовали, но они были слишком дорогими, и большинство компьютеров содержало только один модуль выполнения команд. Чтобы добиться параллельного выполнения, процессор просто очень быстро переключался между задачами, поочередно выполняя их в соответствии с установленными приоритетами. Таким образом, несколько задач могли выполняться как бы одновременно, но не параллельно.

В настоящее время появились многоядерные процессоры, в которых на одном кристалле располагаются несколько ядер, способных параллельно выполнять задачи. Помимо этого, постепенно набирают популярность и многопроцессорные системы, что позволит значительно поднять производительность компьютеров.

12.1. Класс *Thread*

Типы и классы, отвечающие за многопоточность, находятся в пространстве имен `System.Threading`. Основным классом в этом пространстве является `Thread`, который как раз и создает поток и предоставляет нам необходимые рычаги управления. Этот класс служит для управления существующими потоками с помощью статических методов и для создания второстепенных потоков.

Мы с вами уже использовали класс `Thread`, а точнее, его статичный метод `Sleep()`. Этот метод останавливает выполнение текущего потока на указанное в качестве параметра количество миллисекунд. Но т. к. мы тогда не создавали никаких дочерних потоков, а работали только в главном потоке, который создается автоматически, то получали лишь задержку главного потока выполнения команд.

Сейчас нас больше интересуют вторичные потоки и процесс их создания. Давайте рассмотрим сразу же пример и параллельно будем знакомиться с теоретической частью. Создайте новое консольное приложение и напишите в нем код из листинга 12.1.

Листинг 12.1. Простой пример работы с потоками

```
class Program
{
    static void Main(string[] args)
    {
        Thread t = new Thread(new ThreadStart(ThreadProc));
        t.Start();
        string s;
        do
        {
            s = Console.ReadLine();
            Console.WriteLine(s);
        } while (s != "q");
    }

    public static void ThreadProc()
    {
        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine("Это поток");
        }
    }
}
```

```
        Thread.Sleep(1000);  
    }  
}  
}
```

Выполнение программы начинается с метода `Main()`, поэтому давайте и мы начнем рассмотрение примера с этого метода. В самом начале создается экземпляр класса `Thread`. Существуют четыре перегруженных конструктора, но наибольший интерес представляют два из них:

```
Thread(ThreadStart)  
Thread(ParameterizedThreadStart)
```

В качестве параметра в обоих случаях конструктор получает переменную делегата. Но что такое *делегат*? Это описание метода, который можно регистрировать в качестве обработчика события. Однако обработчиками событий возможности делегатов не ограничиваются. В нашем случае делегаты `ThreadStart` и `ParameterizedThreadStart` описывают, как должен выглядеть метод, который будет запущен в отдельном потоке, параллельно основному потоку программы.

Теперь посмотрим на разницу между этими двумя делегатами:

- ❑ `ThreadStart` указывает на то, что метод не должен ничего возвращать и не должен ничего получать. Именно таким и является метод `ThreadProc()` в нашем примере. Правда, наш метод потока объявлен как статичный, но в общем случае это не обязательно. В нашем примере статичность необходима, потому что у приложения нет объектов и мы работаем в статичном методе `Main()`, из которого можно обращаться только к статичным свойствам и методам;
- ❑ `ParameterizedThreadStart` определяет метод, который не должен ничего возвращать, но может получать один параметр типа `Object`. Сразу возникает вопрос: а что если нам нужно передать в поток сразу несколько параметров? Никто не запрещает вам оформить эти параметры в виде объекта или структуры и передать такой объект, который будет содержать множество значений. Если параметры однотипные, то их можно оформить в виде массива.

Итак, в нашем примере мы создаем новый объект потока `t` и в качестве параметра передаем конструктору в виде делегата `ThreadStart` метод `ThreadProc()`, который и будет запущен в отдельном потоке. Именно будет, потому что простого создания потока недостаточно. Чтобы поток начал свое выполнение, нужно вызвать его метод `Start()`. Именно это мы и делаем во второй строке.

Теперь нам нужно убедиться в параллельности выполнения двух потоков: основного, который выполняет на начальном этапе метод `Main()`, и вторичного, который начал выполнение метода `ThreadProc()`. Для этого в методе `Main()` создается цикл, который ожидает ввода со стороны пользователя и будет выполняться до тех пор, пока пользователь не введет букву `q`. В это время в методе `ThreadProc()` запускается цикл из 10 шагов, в котором каждую секунду в консоль выводится сообщение. Чтобы сделать задержку потока на 1 секунду, используется статичный метод `Sleep()` класса `Thread`.

Запустите приложение и попробуйте что-нибудь вводить в консоль. Обратите внимание, что в процессе вашего ввода может неожиданно появиться сообщение от потока. Это говорит о том, что у нас действительно есть два потока.

Попробуйте теперь запустить еще раз приложение и ввести букву `q`, чтобы цикл в основном потоке прервался. Обратите внимание, что приложение не закрылось, — сообщения от вторичного потока все еще идут. Выполнение программы происходит до тех пор, пока основной поток не завершит выполнение всех команд. Но основной поток может не завершить свою работу, если есть дочерние потоки, выполняющиеся на переднем плане. Метод `Main()` после прочтения буквы `q` завершил работу, и подтверждением этого является то, что мы больше ничего не можем ввести, и никто не ожидает нашего ввода за консолью. Почему же программа не завершается, а консоль закрывается только по завершении работы вторичного потока, когда он выведет все свои 10 сообщений? Секрет кроется в свойстве `IsBackground` потока.

По умолчанию все потоки создаются как потоки переднего плана, и у них свойство `IsBackground` равно `false`. Процесс не может завершиться, пока у него есть работающие вторичные потоки переднего плана. Если вы не хотите, чтобы поток блокировал завершение вашего приложения, то следует изменить свойство `IsBackground` на `true`. Попробуйте сейчас сделать это для нашего примера до вызова метода `Start()`. Запустите приложение и введите букву `q`. Приложение завершится сразу, вне зависимости от того, успел ли вторичный поток обработать все свои 10 шагов.

Давайте посмотрим, какие еще свойства предлагает нам класс `Thread`:

- ☐ `IsAlive` — свойство равно `true`, если поток сейчас запущен;
- ☐ `Name` — здесь вы можете указать дружественное имя;
- ☐ `Priority` — через это свойство можно изменить приоритет выполнения потока. От приоритета зависит, сколько процессорного времени будет выделено потоку;
- ☐ `ThreadState` — состояние потока. Поток может находиться в следующих состояниях:
 - `Running` — поток выполняется;
 - `StopRequested` — запрошена остановка потока;
 - `SuspendRequested` — запрошена приостановка потока;
 - `AbortRequested` — запрошена операция прерывания выполнения;
 - `Background` — поток выполняется в фоне;
 - `Unstarted` — поток еще не выполнялся (не вызывался метод `Start()`);
 - `Stopped` — поток остановлен;
 - `Suspended` — поток приостановлен;
 - `Aborted` — выполнение прервано;
 - `WaitSleepJoin` — поток заблокирован.

Наиболее интересным свойством является приоритет выполнения потока. Он имеет тип данных перечисления `ThreadPriority`, которое позволяет вам указывать следующие значения:

- ☐ `Lowest` — самый низкий приоритет;
- ☐ `BelowNormal` — ниже нормального;
- ☐ `Normal` — нормальный приоритет (значение по умолчанию);
- ☐ `AboveNormal` — выше нормального;
- ☐ `Highest` — наивысший приоритет.

Изменение приоритета потока не может сильно повлиять на выделяемое процессорное время. Это всего лишь ваши запросы, а что реально будет выделено процессу — зависит от ОС. По умолчанию все потоки получают приоритет `Normal`. Указав значение `Highest`, вы всего лишь просите у ОС давать вам больше процессорного времени по сравнению с другими потоками. Но нет гарантии, что ОС станет уделять потоку, объявленному высокоприоритетным, лишнее внимание.

В связи с этим в большинстве случаев свойство приоритета оставляют по умолчанию. Да и изменять его, как правило, не имеет смысла. Чаще всего мне приходилось понижать приоритет потока, когда я создавал какой-то вспомогательный поток, который должен выполнять обслуживающие операции в фоне. Поэтому, чтобы поток сильно не отбирал процессорное время у основного потока, приоритет сервисного лучше понизить.

Методы у класса `Thread` не менее интересны и полезны:

- ☐ `Abort()` — заставляет систему прервать поток. Он не будет прерван мгновенно, система только постарается как можно скорее прервать его работу;
- ☐ `Interrupt()` — прервать поток, который находится в состоянии `WaitSleepJoin`;
- ☐ `Join()` — заблокировать текущий поток, пока не завершит работу поток, указанный в качестве параметра;
- ☐ `Resume()` — возобновить работу потока, который был приостановлен;
- ☐ `Suspend()` — приостановить выполнение потока.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter12\ThreadTest` сопровождающего книгу электронного архива (см. приложение).

12.2. Передача параметра в поток

В тех случаях, когда необходимо в метод потока передать какое-либо значение (а такое бывает очень часто), можно использовать параметризованный вариант делегата: `ParameterizedThreadStart`. В листинге 12.2 показан модифицированный код, в котором потоку передается в качестве параметра количество шагов, которые он должен сделать в цикле параллельно основному потоку.

Листинг 12.2. Передача параметра в поток

```
static void Main(string[] args)
{
    Thread t = new Thread(new ParameterizedThreadStart(ThreadProc));
    t.IsBackground = true;
    t.Start(5);
    ...
}
// метод, выполняемый в потоке
public static void ThreadProc(Object number)
{
    int loop_number = (int)number;
    for (int i = 0; i < loop_number; i++)
    {
        Console.WriteLine("Это поток");
        Thread.Sleep(1000);
    }
}
```

На этот раз конструктору класса `Thread` передается делегат `ParameterizedThreadStart`, который определяет метод, получающий параметр типа `Object`. Для этого пришлось добавить этот параметр методу `ThreadProc()`. Само значение передается через метод `Start()`, который запускает поток на выполнение.

Благодаря универсальности типа данных `Object` мы можем передать методу любые данные, даже простой тип `int`. Простой тип данных, такой как число, будет упакован в объект класса `IntXX` (`XX` — разрядность числа) и передан методу потока. Например, в нашем случае число 5, скорее всего, будет передано в виде типа данных `Int32`.

Тут я хочу заметить, что метод `Start()`, который должен запускать поток на выполнение, на самом деле только информирует систему о том, что мы хотим запустить поток. Нет никакой гарантии, что поток запустится мгновенно и именно параллельно основному процессу. Когда этот поток будет в реальности запущен, известно только одной операционной системе. К чему я это? Когда будете писать свой код потоков, не стоит надеяться, что метод потока начнет свое выполнение до выполнения первого оператора, следующего за `Start()`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter12\ParamThreadTest` сопровождающего книгу электронного архива (см. приложение).

12.3. Конкурентный доступ

Когда несколько потоков обращаются к одному и тому же ресурсу, то между ними возникает конкуренция. Каждый поток пытается получить доступ к ресурсу первым, и система может давать этот доступ в хаотичном порядке. Давайте посмотрим на пример из листинга 12.3.

Листинг 12.3. Код потоков с конкурентным доступом к данным

```
class ThreadTester
{
    public ThreadTester()
    {
        for (int i = 0; i < 5; i++)
        {
            Thread t = new Thread(new ThreadStart(ThreadFunc));
            t.Name = "Поток " + i.ToString();
            t.Start();
        }
        Console.ReadLine();
    }

    // метод, который будем выполнять в потоке
    void ThreadFunc()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(Thread.CurrentThread.Name + " - " +
                               i.ToString());
            Thread.Sleep(100);
        }
    }
}
```

В этом примере конструктор класса `ThreadTester` создает 5 потоков, каждый из которых выводит в консоль по 5 чисел с задержкой в 100 миллисекунд. Для удобства потокам при создании даются имена, а внутри метода потока, чтобы получить его имя, я использую конструкцию `Thread.CurrentThread.Name`.

Цикл создания потоков выполняется достаточно быстро, и на моем компьютере получилось так, что задержка в 100 миллисекунд для всех потоков завершалась примерно в одно и то же время, поэтому в этот момент в консоли начинался бардак. Кто первый встал, того и тапки, — поэтому числа в консоли появились абсолютно хаотично и без четкой последовательности.

А как сделать, чтобы каждый из потоков, если это нужно, отработывался бы индивидуально и их числа шли бы строго последовательно? То есть пока выполняется

метод `ThreadFunc()` для одного потока, никакой другой поток не мог бы получить доступ к этому же ресурсу? Да легко! Нужно код, который должен вызываться для каждого потока в отдельности, заключить в блок `lock`:

```
void ThreadFunc()
{
    lock (this)
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine(Thread.CurrentThread.Name + " - " +
                               i.ToString());
            Thread.Sleep(100);
        }
    }
}
```

В качестве параметра ключевое слово `lock` получает объект-маркер, используемый для синхронизации. Самый простой способ передать потоку объект — использовать ключевое слово `this`, в котором находится текущий объект.

Если теперь запустить пример, то пока один объект выводит данные в консоль, а точнее, выполняет код из блока `lock`, ни один другой поток в код этого блока не войдет, поэтому результаты выводов потоков будут идти последовательно.

Ключевое слово `lock` введено для вашего удобства. На самом деле при компиляции это слово заменяется на следующий код с использованием класса `Monitor`:

```
Monitor.Enter(this);
try
{
    for (int i = 0; i < 5; i++)
    {
        Console.WriteLine(Thread.CurrentThread.Name + " - " +
                           i.ToString());
        Thread.Sleep(100);
    }
}
finally
{
    Monitor.Exit(this);
}
```

Класс `Monitor` содержит два статических метода, которые и используются для создания синхронизации: метод `Enter()` входит в код синхронизации, а метод `Exit()` — выходит.

Чтобы лучше понять проблему конкуренции доступа, давайте рассмотрим пример. Допустим, у вас есть два потока. Первый поток подготавливает список файлов в определенном каталоге, а другой поток отображает эти данные на экране. Если

первый поток не успеет сформировать список нужных файлов, то второй отобразит некорректную информацию. Получается, что при обращении к одним и тем же данным со стороны разных потоков мы можем встретиться с проблемой некорректности данных.

На самом деле даже такие операции, как присвоение или простые математические, не являются целостными и тоже конкурируют за доступ к переменным. Чтобы решить эту проблему, можно весь код заключать в блок `lock`, но поддержка такой синхронизации достаточно накладно сказывается на процессоре и отнимает лишние ресурсы.

Проблему решает класс `Interlocked`, который имеется в пространстве имен `System.Threading`. У этого класса есть несколько статичных членов, которые гарантируют целостность (атомарность) выполняемой операции:

- ❑ `Add()` — складывает два числа, а результат возвращает в первом параметре;
- ❑ `CompareExchange()` — сравнивает два значения, и если они равны, то заменяет первое из сравниваемых значений;
- ❑ `Decrement()` — безопасно уменьшает на единицу;
- ❑ `Increment()` — безопасно увеличивает на единицу;
- ❑ `Exchange()` — безопасно меняет два значения местами.

А что если вам нужно синхронизировать все методы и свойства целого класса? Тогда идеальным вариантом будет использование атрибута `Synchronization`. Поставив этот атрибут перед объявлением класса, вы защитите все члены класса:

```
[Synchronization]
public class MyClass: Объект
{
    // методы и свойства класса
}
```

Чтобы использовать этот атрибут, нужно подключить к модулю пространство имен `System.Runtime.Remoting.Contexts`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter12\ConcurrentAccess` сопровождающего книгу электронного архива (см. приложение).

12.4. Пул потоков

На первый взгляд может показаться, что создание потоков связано с лишними затратами — ведь нужно выделить для нового потока выполнения команд какие-то ресурсы. Да, ресурсы нужны, но все не так уж и страшно. Для повышения производительности система использует *пул* (набор) потоков, который позволяет как раз повторно использовать ресурсы потоков.

Для управления пулом в .NET есть класс `ThreadPool`. Вы помещаете методы в пул на выполнение в потоке, и как только какой-то поток в пуле освободится, он (освободившийся поток) будет выполнен. Чтобы поместить что-то в очередь пула, используется метод `QueueUserWorkItem()`. Есть несколько перегруженных методов, но наиболее интересным является вариант, который получает два параметра: делегат `WaitCallback` и объектную переменную `state`. Делегат `WaitCallback` определяет метод следующего вида:

```
public delegate void WaitCallback(  
    Object state  
)
```

Обратите внимание на имя параметра делегата. Оно такое же, как и имя второго параметра `WaitCallback`. И это не случайно. То, что мы укажем во втором параметре `WaitCallback`, будет передано в качестве единственного параметра делегату.

Давайте напишем пример, в котором программа будет рассчитывать в потоке факториалы чисел от 1 до 10. Код такого примера показан в листинге 12.5.

Листинг 12.4. Использование пула потоков

```
class Program  
{  
    static void Main(string[] args)  
    {  
        WaitCallback callback = new WaitCallback(FactFunc);  
        // цикл помещения делегатов в очередь пула  
        for (int i = 1; i < 10; i++)  
        {  
            ThreadPool.QueueUserWorkItem(callback, i);  
        }  
        Console.ReadLine();  
    }  
  
    // делегат расчета факториала  
    static void FactFunc(Object state)  
    {  
        int num = (int)state;  
        int result = 1;  
        for (int i = 2; i < num; i++)  
            result *= i;  
        Console.WriteLine(result);  
    }  
}
```

В методе `Main()` мы сначала создаем экземпляр делегата `WaitCallback`. Потом запускается цикл, внутри которого в пул добавляются делегаты, в качестве параметра которым передается значение переменной `i`. Наполнив пул, система будет брать

готовые объекты потоков и использовать их для выполнения кода нашего делегата. Если пул будет меньше созданных делегатов, то по мере выполнения и освобождения потоков пула будут выполняться остальные делегаты.

При работе с пулом нужно учитывать, что все его потоки являются фоновыми и выполняются с нормальным приоритетом (`ThreadPriority.Normal`).

Давайте рассмотрим, какие еще сервисные методы предлагает нам класс `ThreadPool`:

- ❑ `GetMaxThreads()` — позволяет узнать максимальное количество потоков, выполняемых одновременно в пуле;
- ❑ `GetAvailableThreads()` — возвращает количество свободных потоков в пуле;
- ❑ `GetMinThreads()` — возвращает минимальное количество подготовленных к выполнению в пуле потоков, которые всегда находятся в режиме ожидания поставки в очередь делегата;
- ❑ `SetMaxThreads()` — позволяет изменить максимальное количество потоков в пуле;
- ❑ `SetMinThreads()` — позволяет изменить минимальное количество потоков в пуле.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter12\ThreadPoolProject` сопровождающего книгу электронного архива (см. приложение).

12.5. Домены приложений .NET

В классических Windows-приложениях для платформы Win32 исполняемый код помещается непосредственно в процесс, в котором и выполняется. Процесс может создавать потоки, которые будут выполняться параллельно с ним. В .NET есть дополнительный промежуточный уровень, называемый *доменом приложения*, или `AppDomain`. Один процесс может состоять из нескольких доменов, каждый из которых может выполнять свой собственный исполняемый файл (сборку).

Для чего был введен этот дополнительный уровень? Во-первых, для обеспечения независимости от платформы. Каждая платформа по-своему работает с процессами, а домены приложения позволяют абстрагироваться от того, как та или иная платформа работает с исполняемым объектом. Вторая причина — это надежность, потому что домены не влияют на работу всего приложения. Если работа одного из доменов нарушена, то приложение в целом продолжает работать и будет выполнять остальные домены.

Домены приложения работают независимо друг от друга и не разделяют никаких данных. Код одного домена не может получить доступ к свойствам и значениям другого. Для обмена информацией придется задействовать внешние хранилища информации, которые могут разделяться (например, файлы), или использовать удаленное взаимодействие или простой сетевой обмен данными.

При создании нового процесса в нем создается домен по умолчанию, в котором и будет выполняться код запущенного процесса. Вы можете создавать дополнитель-

ные домены или управлять уже существующими. Несмотря на то что такая необходимость возникает очень редко, эту тему следует рассмотреть хотя бы в общеобразовательных целях.

Для работы с доменом приложения в .NET есть класс `AppDomain`, который находится в пространстве имен `System`. Используя его статические методы, вы можете управлять доменами процесса. У этого класса есть одно статическое свойство — `CurrentDomain`, которое хранит текущий домен и два статических метода, которые могут нас заинтересовать:

- ❑ `CreateDomain()` — создать новый домен в текущем процессе;
- ❑ `Unload()` — выгружает указанный в качестве параметра домен.

Существуют еще три статических метода, но один из них (`GetCurrentThreadId()`) не поддерживается — он устарел и остался только для совместимости, а два других (`Equals()` и `ReferenceEquals()`) предназначены для сравнения объектов и ссылок.

Давайте напишем один очень интересный пример, который покажет нам домен на практике. Точнее сказать, нам придется написать целых два приложения, и оба будут консольными для простоты эксперимента.

Первый проект назовем `DomainTest` и в его методе `Main()` напишем следующий код:

```
static void Main(string[] args)
{
    Console.WriteLine("Это внешняя сборка");
    Console.WriteLine(AppDomain.CurrentDomain.FriendlyName);
    Console.ReadLine();
    Console.WriteLine("Завершаем работу");
}
```

Здесь выводится в консоль приветственное сообщение, имя текущего домена, запрашивается у пользователя ввод и выводится прощальное сообщение. Ничего сложного — просто идентификация того, что перед нами находится определенная сборка. Для определения текущего дружественного имени домена обращаемся к свойству `AppDomain.CurrentDomain.FriendlyName`.

Скомпилируйте проект, чтобы создать сборку, и выполните ее. При запуске исполняемого файла система создаст процесс, в котором будет создан домен по умолчанию, внутри которого и станет происходить выполнение исполняемого кода. Всего этого мы не видим, оно скрыто от нашего взгляда. Домены по умолчанию в качестве дружественного имени получают имя текущего исполняемого файла. В нашем случае исполняемый файл `DomainTest.exe`, и вы именно это имя должны увидеть на экране. Если вы запускаете файл из Visual Studio в режиме отладки, то имя домена может быть `DomainTest.vshost.exe`. Если заглянуть в папку `bin\Debug`, то вы увидите там файл с таким именем, и именно в домене этой сборки выполняется наш код, запущенный в режиме отладки.

Теперь создаем новое консольное приложение, которое я назвал `AppDomainProject`, а в его методе `Main()` пишем следующий код:

```
static void Main(string[] args)
{
    Console.WriteLine("Сейчас будем запускать другую сборку в нашем процессе");
    AppDomain ad = AppDomain.CreateDomain("Мой домен");
    ad.ExecuteAssembly(
        @"F:\Source\Chapter12\DomainTest\bin\Release\DomainTest.exe");
    Console.ReadLine();
}
```

Вот тут кроется самое интересное. После приветственного сообщения вызываем статичный метод `CreateDomain()` для создания нового домена внутри текущего процесса. Методу `CreateDomain()` передается дружественное имя домена, которое будет назначено ему, а в качестве результата мы получим объект класса `AppDomain`.

Следующим шагом запускаем сборку на выполнение. Причем именно ту, которую мы создали ранее: `DomainTest.exe`. Это делается с помощью метода `ExecuteAssembly()` домена. В качестве параметра метод получает путь к сборке, которую нужно выполнить. На этот раз сборка `DomainTest.exe` будет запущена не в отдельном процессе и домене по умолчанию, а в текущем процессе и созданном нами домене.

Чтобы приложение не завершило работу, в последней строке вызываем метод `ReadLine()`.

Запустите приложение на выполнение. На этот раз после приветствия, которое есть в коде нашей сборки, появляется приветствие внешней сборки, которую мы запустили в текущем процессе, и появляется имя домена. Имя домена на этот раз "Мой домен". Так как все выполняется в одном процессе, то обе сборки работают с одной и той же консолью, и весь вывод идет в одно окно консоли.

Нажмите клавишу `<Enter>`, и вы увидите прощальное сообщение сборки `DomainTest.exe`. То есть после создания домена именно этот домен получил управление консолью, и он ожидал ввода. Но процесс не завершил работу, потому что ввода ожидает домен по умолчанию, в котором работает сборка `AppDomainProject.exe`. Завершил работу только домен, который мы создавали явно. Результат работы примера можно увидеть на рис. 12.1.

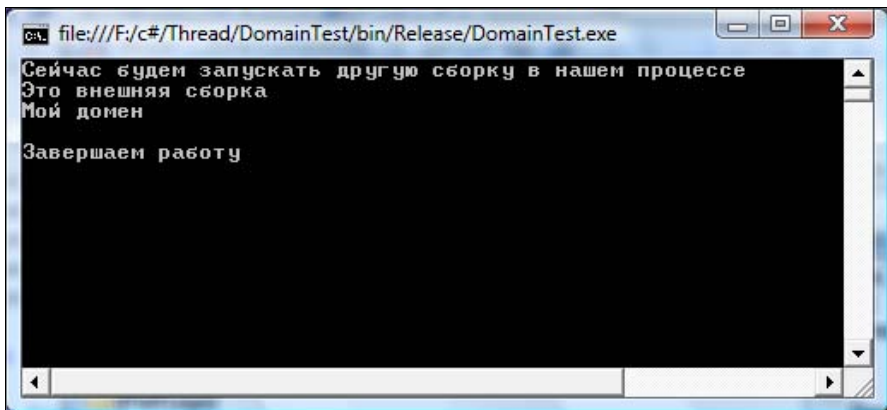


Рис. 12.1. Результат работы программы

ПРИМЕЧАНИЕ

Исходный код примеров к этому разделу можно найти в папках *Source\Chapter12\AppDomainProject* и *Source\Chapter12\DomainTest* сопровождающего книгу электронного архива (см. приложение).

12.6. Ключевые слова *async* и *await*

Архитектура Windows существует уже многие годы, и изначально потоки были как бы исключительной ситуацией, и их создавали только в крайнем случае. Мне кажется, все изменилось после появления iOS, когда Apple показала, что интерфейс должен отзываться на действия пользователя, невзирая ни на что.

Потоки действительно делают код более гибким, и с ними проще создавать интерфейс, который не будет блокироваться в ожидании завершения долго выполняющихся задач.

Чтобы сделать для программистов Windows-платформы разработку потоков проще, Microsoft разработала новый подход, основанный на применении двух новых ключевых слов: *async* и *await*.

Исследуем все с самого начала на классическом и легко понятном примере работы с Интернетом. Доступ к Интернету не всегда быстрый и далеко не самый стабильный. На мобильных устройствах — если у вас нет скоростного LTE-подключения — эта проблема еще серьезнее. Если вызвать методы доступа к сайту синхронно, то основной поток остановится в ожидании медленного ответа с сервера, и окно программы даже не станет обновляться. Проблему можно решить с помощью потоков, но с точки зрения программирования это не самый простой и удобный подход. Задачи предлагают нам более элегантное решение.

Для примера я создал класс *AsyncSampleClass* с одним методом, который загружает с моего сайта файл *robots.txt*.

Листинг 12.5. Потоки с использованием *async/await*

```
class AsyncSampleClass
{
    public async Task<string> AccessTheWebAsync() {
        HttpClient client = new HttpClient();

        Task<string> getStringTask =
            client.GetStringAsync("http://www.flenov.info/robots.txt");

        return await getStringTask;
    }
}
```

В этом методе создается новый экземпляр класса *HttpClient*, который умеет работать со страницами в Интернете. Потом вызывается метод *GetStringAsync*, в качестве параметра для которого указан адрес (URL) к файлу *robots.txt* на моем сайте.

С помощью таких файлов сайты обычно сообщают, какие страницы поисковые системы могут индексировать, а какие — нет.

Выполнение этого метода создает задачу `Task`, которая направляет на сервер запрос на загрузку файла и вместо блокировки текущего потока нам возвращается задача, через которую мы потом можем получить доступ к результату, когда он будет готов.

Если ваше приложение с визуальным интерфейсом, то интерфейс не будет блокироваться, потому что загрузка файла идет асинхронно.

ПРИМЕЧАНИЕ

Исходный код консольного примера использования класса `HttpClient` можно найти в папке `Source\Chapter12\TaskSample` сопровождающего книгу электронного архива (см. приложение).

В листинге 12.5 показаны задачи, которые требуют подключения пространства имен `System.Threading.Tasks` и клиента для работы с HTTP-протоколом, которому нужно пространство имен `System.Net.Http`. Их следует подключить как обычно:

```
using System.Net.Http;
using System.Threading.Tasks;
```

Теперь посмотрим на код, который использует класс из листинга 12.5 (листинг 12.6).

Листинг 12.6. Использование асинхронного метода

```
// создать класс
AsyncSampleClass c = new AsyncSampleClass();

// вызвать метод загрузки файла из Интернета
Task<string> asyncContent = c.AccessTheWebAsync();

// показать состояние задачи
Console.WriteLine("Состояние: " + asyncContent.Status);

// дождаться загрузки выполнения
string webContent = asyncContent.Result;

// еще раз отобразить статус
Console.WriteLine("Состояние: " + asyncContent.Status);

// отобразить содержимое файла
Console.WriteLine("Результат с сайта:");
Console.WriteLine(webContent);
Console.ReadLine();
```

Мы здесь вызываем асинхронный метод загрузки файла и сохраняем результат в задаче. Эта задача выполняется не мгновенно, поэтому результата пока, скорее

всего, получено не будет. Точнее сказать, задача была только создана, но еще не начала выполняться. У этой задачи есть свойство `Status`, через которое можно проверить текущее состояние, и в настоящий момент оно будет равно `WaitingForActivation`.

Через объект задачи `asyncContent` мы можем посмотреть статус выполнения, а также через свойство `Result` задачи обратиться к результату:

```
string webContent = asyncContent.Result;
```

При обращении к свойству `Result` мы блокируем текущий поток в ожидании результата. Согласно документации это эквивалент вызова метода `Wait` и ожидания загрузки файла:

```
asyncContent.Wait();
```

Если во время выполнения задачи произойдет ошибка, а мы будем обращаться к результату через свойство `Result`, то увидим исключительную ситуацию `AggregateException`. Это плохо, потому что мы не будем знать, что реально привело к проблеме.

Чтобы в случае ошибки увидеть реальную исключительную ситуацию, рекомендуется использовать `GetAwaiter().GetResult()`:

```
asyncContent.GetAwaiter().GetResult()
```

Здесь сначала вызывается метод `GetAwaiter` объекта задачи, который позволяет дождаться отработки выполнения потока. Метод вернет нам объект класса `TaskAwaiter`, у которого есть метод `GetResult`, позволяющий получить результат.

В любом случае, какой бы вы способ ни выбрали, в этот момент выполнение программы остановится в ожидании результата.

Когда результат получен, я еще раз вывожу статус, и на этот раз он должен быть `RanToCompletion`, потому что поток обязан был завершить работу.

Полный результат выполнения программы будет таким:

```
Состояние: WaitingForActivation
Состояние: RanToCompletion
Результат с сайта:
User-agent: *
Allow: /
Host: www.flenov.info
```

Я начал с этого простого сетевого примера, и в нем задача, которая выполняется в отдельном потоке, создается методом `GetStringAsync` класса `HttpClient`. То есть мы рассмотрели случай, как работать с кодом, когда какой-то существующий API возвращает нам не реальный результат, а задачу, которая по завершении работы в потоке позволит нам узнать результат.

Ожидание результата с помощью `Result` и `GetAwaiter().GetResult()` чревато серьезной проблемой — это может привести к «мертвой» блокировке. Поясню, что это значит. Допустим, два потока создают задачи, которые не должны блокировать

выполнение. Да, каждый поток может иметь свои задачи. То есть оба потока сказали, что они находятся в режиме ожидания, и дали процессору шанс выполнять другие задачи. Затем они оба завершили свою работу и пытаются вернуться обратно в свой оригинальный поток, но если при этом они заблокируют друг друга, то такое состояние и называется «мертвой» блокировкой.

Вот максимально упрощенный пример, который приводит к блокировке:

```
Поток 1 выполняет код 1
Поток 2 выполняет код 2
Поток 3 выполняет код 3
Поток 1 создает задачу для кода 1
Поток 4 начинает выполнять код 1
Поток 1 свободен, пока ждет потока 4
Поток 2 создает задачу для кода 2
Поток 1 свободен и берет выполнение кода 2
Поток 3 создает задачу для кода 3
Поток 2 свободен и берет на себя код 3
```

И вот мы подобрались к проблеме. Поток 3 выполняет код 2, а поток 2 выполняет код 3. Они выполняют код друг друга, и когда поток 2 закончит выполнять код 3, он должен вернуть код 3 потоку 3, где он работал изначально, но не сможет этого сделать, потому что поток 3 должен вернуть код 2 потоку 2.

Вы можете подумать, что проблему можно решить банальным обменом — два потока меняются веткой кода, и проблема решена, но в реальной жизни все может быть более сложно: поток 1 зависит от 2, поток 2 зависит от 3, поток 3 зависит от 4, а четыре от 1. Распутать такую циклическую цепочку будет уже не так просто.

Это проблема, и тут есть два варианта решения. Идеальный вариант — не ожидать результата через свойство `Result` и метод `GetAwaiter().GetResult()`, а вместо этого все методы должны возвращать задачу.

В идеале если метод возвращает задачу, то вы должны как бы передавать ее дальше. Теоретически мы должны были бы написать код таким образом:

```
async Task<int> Main() {
    AsyncSampleClass c = new AsyncSampleClass();
    string result = await c.AccessTheWebAsync();
}
```

Метод `AccessTheWebAsync` возвращает задачу `Task`, но вместо того, чтобы сохранять задачу и ожидать потом результат, примененный мною новый оператор `await` ожидает результат и дает нам уже не задачу, а строку. Проблема в том, что если мы используем этот подход, то и метод `Main` должен возвращать задачу, что я и попытался сделать в приведенном примере (добавил в объявление метода оператор `async`, чего для `Main` делать нельзя). Так что рекомендуется использовать показанный подход повсеместно, за исключением случаев с методом `Main`, просто потому, что он с ним не работает. Вот когда мы займемся веб-программированием и станем рассматривать контейнеры, то там этот подход работать будет. Сейчас же я только

хотел показать идею — как рекомендуется делать в тех случаях, когда применение показанного подхода возможно.

Второе решение: попросить .NET не менять контекст (поток) по завершении работы — для этого нужно после задачи `ConfigureAwait(false)` указать:

```
string webContent =  
    asyncContent.ConfigureAwait(false).GetAwaiter().GetResult();
```

Этот подход позволяет избежать блокировок, потому что `ConfigureAwait(false)` говорит: не возвращайся в оригинальный контекст и продолжай выполняться в том же потоке, где и закончил выполнение. Очень часто это нормально — продолжать выполнение и не менять контекст.

Нам еще нужно посмотреть, как самим создать задачу, которая будет выполнять наш собственный код.

Представим, что расчет факториала — это очень медленная и сложная операция, и чтобы сделать ее такой медленной, можно просто добавить задержку на каждом этапе цикла:

```
for (int i = 1; i <= value; i++) {  
    result *= i;  
    System.Threading.Tasks.Task.Delay(10).Wait();  
}
```

Теперь этот код будет выполняться намного дольше. Если вы хотите вычислить факториал числа 10, то за счет задержек код будет выполняться 100 миллисекунд, что вполне заметно на глаз. Можно код затормозить еще сильнее, но для нашего примера этого будет достаточно.

Итак, нам нужно создать задачу, которая будет выполняться параллельно с нашим кодом, и для этого можно использовать статичный метод `Run` класса `Task`:

```
public Task<int> LongRunningFactorial(int value)  
{  
    return Task.Run(() =>  
    {  
        int result = 1;  
        for (int i = 1; i <= value; i++) {  
            result *= i;  
            System.Threading.Tasks.Task.Delay(10).Wait();  
        }  
        return result;  
    }));  
}
```

В качестве параметра метод `Run` получает `Action`, которые легко создаются с помощью *анонимных функций* или `Lambda Expression` (лямбда-выражений). Когда мы работали с LINQ, то как раз писали подобные лямбда-выражения, — тогда они выглядели так:

```
m => код, использующий m
```

Сейчас мы делаем почти то же самое, только вместо переменной `m` у нас круглые скобки. А когда мы используем круглые скобки? При объявлении методов, а значит, вместо переменной `m` определенного типа мы работаем с методом или, точнее, с функцией. Отличие функции от метода заключается в том, что метод принадлежит какому-то классу, а функция существует сама по себе и не принадлежит никому. В старых языках программирования можно было создавать функции типа:

```
void Factorial(int value) {
    . . .
}
```

без указания класса, в том числе так могли создаваться глобальные функции. В .NET такое запрещено — все функции должны принадлежать классам, а значит, они становятся методами.

У нашей функции просто нет имени, поэтому и говорят: *анонимная функция*. Она не имеет ни имени, ни класса:

```
() => {
    // код анонимной функции
}
```

Именно такую анонимную функцию мы должны передать методу `Run`. Метод создаст задачу и вернет ее в качестве результата.

Теперь метод `LongRunningFactorial` работает примерно так же, как и метод загрузки данных из Интернета, — создает задачу и, не дожидаясь окончания ее выполнения, разрешает нам продолжить работу.

Теперь посмотрим, как этот метод можно использовать:

```
AsyncSampleClass2 sample = new AsyncSampleClass2();
var factorialTask = sample.LongRunningFactorial(12);
while (factorialTask.Status != TaskStatus.RanToCompletion) {
    Console.WriteLine("Статус = " + factorialTask.Status);
    System.Threading.Tasks.Task.Delay(100).Wait();
}
Console.WriteLine("Статус = " + factorialTask.Status);
Console.WriteLine("Результат = " + factorialTask.Result);
```

В этом примере создается класс `AsyncSampleClass2`, внутри которого я реализовал метод `LongRunningFactorial`, и вызываю этот метод, чтобы создать задачу.

Там запускается цикл `while`, который проверяет статус задачи. Если статус не равен `TaskStatus.RanToCompletion`, то мы отображаем текущий статус и ждем 100 миллисекунд, чтобы дать возможность задаче закончить свои расчеты. Когда статус изменяется на `TaskStatus.RanToCompletion`, отображаем результат и выходим.

Результат выполнения этого примера будет таким:

```
Статус = WaitingToRun
Статус = Running
```



```
Статус = RanToCompletion
```

```
Результат = 479001600
```

ВНИМАНИЕ!

Я использую цикл `while` только для того, чтобы показать вам, как меняется статус задачи. В реальных примерах вы не должны таким образом ожидать окончания задачи — дожидаться выполнения задачи нужно следующим образом:

```
factorialTask.GetAwaiter().GetResult()
```

Моей целью было показать вам, как работают потоки, и дать по этому вопросу начальные базовые знания. Используйте потоки и задачи для всех долго выполняющихся задач и тестируйте интерфейс программ, чтобы убедиться, что интерфейс не блокируется и всегда отзывается на действия пользователей.

Если метод выполняет что-то асинхронно, то он обязательно должен возвращать задачу. Следующий код будет некорректным:

```
public async void AccessTheWebAsync() {  
    await DoSomething();  
}
```

Внутри метода вызывается какая-то асинхронная операция, но метод ничего не возвращает, и это приведет к тому, что платформа не будет иметь якоря, который позволил бы дождаться ожидания. Если метод должен возвращать что-то, то этот тип данных нужно обрамлять в `Task`. Например, если метод должен возвращать число, то его нужно оформить так:

```
public async Task<int> AccessTheWebAsync()
```

Если метод ничего не должен возвращать, то возвращаем `Task`:

```
public async Task AccessTheWebAsync()
```

Вернемся к примеру, когда в консольном приложении я вызывал из метода `Main` другой асинхронный метод:

```
async Task<int> Main() {  
    AsyncSampleClass c = new AsyncSampleClass();  
    string result = await c.AccessTheWebAsync();  
}
```

Если внутри метода есть вызов `await`, то мы должны вернуть задачу, и этот процесс рекурсивный. Посмотрим на следующий гипотетический пример:

```
async Task<int> Main() {  
    string result = await AsyncMethod1();  
}  
  
async Task AsyncMethod1() {  
    await AsyncMethod2();  
}  
  
async Task AsyncMethod2() {  
    // асинхронная операция  
}
```

Метод `AsyncMethod2` возвращает задачу. Когда мы вызываем этот метод из `AsyncMethod1`, то должны дожидаться этой задачи:

```
await AsyncMethod2();
```

Раз мы ожидаем задачи, значит, `AsyncMethod1` должен возвращать `Task`.

Метод `AsyncMethod1` вызывается из метода `Main`, и здесь снова нужно использовать `await` и возвращать `Task`.

Если где-то в этой цепочке забыть `await`, то асинхронная задача начнет выполняться параллельно основной. Если основной поток завершится раньше, чем параллельный, то параллельный будет остановлен и может не завершить свою работу.

Посмотрим на следующий код:

```
int Main() {  
    await SendEmail();  
}
```

Здесь я инициализирую отправку почты асинхронно и продолжаю выполнение программы. Метод не возвращает задачу, но при этом он может быть абсолютно корректным с точки зрения компилятора. А вот с точки зрения логики он неверный. Дело в том, что если вызвать `SendEmail` параллельно и не вернуть задачу, то никто не будет дожидаться завершения задачи. Когда выполнение метода `Main` завершится, все ресурсы будут освобождены, и если асинхронный поток не успеет к этому времени отправить письмо, то этого уже не произойдет.

Мы обязаны вернуть задачу, чтобы платформа дождалась ее выполнения перед освобождением ресурсов:

```
Task<int> Main() {  
    await SendEmail();  
}
```

12.7. Задачи или потоки — что выбрать?

В этой главе мы рассмотрели сразу две разные концепции: потоки `Thread` и задачи `Task`. Что выбрать для вашего приложения? Потоки — это концепция ОС, когда мы говорим, что код должен выполняться в отдельном потоке. Задачи — это больше абстракция, которая позволяет более эффективно использовать процессор.

Рассмотрим пример веб-запроса. Когда мы в браузере вводим какой-либо интернет-адрес (URL), то запрос идет на сервер, где для обработки нашего запроса выделяется поток, который может обращаться к базе данных и потом возвращать нам результат. Допустим, что у нас есть вот такой код, где `FindBlogItems` обращается к базе данных и ищет там нужные данные:

```
void method() {  
    Blog blog = new Blog();  
    BlogItems items = blog.FindBlogItems();  
}
```

```
        return render(items)
    }
```

Когда мы запрашиваем данные из базы, код программы замирает на `FindBlogItems` — в этот момент идет запрос на сервер, и наш поток продолжит выполнение, когда база данных найдет нужный нам результат. Это проблема — текущий поток блокируется и ничего не делает.

Мы можем создать отдельный поток `Thread` и сказать: давайте в этом потоке будем выполнять запрос, а пока он выполняется, делать что-либо другое. И будет отлично, если нам есть чем заняться, пока второй поток ищет данные в базе. А если нам делать нечего? Придется сидеть и все равно ждать...

Перепишем наш пример с помощью задач:

```
void async method() {
    Blog blog = new Blog();
    BlogItems items = await blog.FindBlogItems();
    return render(items)
}
```

Теперь мы не создаем новый поток, а используем задачи. Но что это значит? Наш поток все также получает запрос от пользователя и начинает выполнение. Когда же запрашиваются данные из базы, то мы создаем задачу, и в этот момент наш поток освобождается и может выполнять любые другие задачи, т. е. не простаивает попусту. Когда же данные от сервера возвращаются, то поток подхватывает результат и продолжает выполнять код.

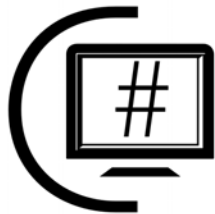
Таким образом, задачи — это абстракция на языке `C#` и платформе, которая лучше и более эффективно использует процессор.

Работу потоков хорошо объяснять на примере повара. Повар — это поток. Когда вы находитесь на кухне и включили чайник, то можете зависнуть в ожидании, пока чайник нагреется, а можете выполнить операцию `await` и заняться другими делами — делать бутерброд. Когда чайник нагреется, то вы можете вернуться к нему и начать заваривать чай.

Если на кухне работают два повара, то это два потока. Если один повар включит чайник, а другой будет делать бутерброды, то работа все еще будет двигаться, но один из поваров просто ждет закипания чайника, и это неэффективно. Можно добавлять поваров, и пусть один смотрит на чайник, другой — на кофемашину, а третий ждет, когда нагреется сковородка. Да, этот подход будет работать, но для поддержания каждого повара нужно платить им зарплату, и точно также ОС нужно тратить ресурсы на поддержание потоков.

Если у вас действительно есть задачи, которые нужно выполнять параллельно, то создавайте потоки и выполняйте их параллельно. Если где-то нужно ожидание окончания работы, то используйте `await` для более эффективного использования ресурсов, чтобы не пришлось нанимать бесконечное количество поваров.

ГЛАВА 13



Добро пожаловать в веб-программирование

Изначально в «Библии C#» я рассказывал про язык на примерах десктопных приложений, но при этом сам с 2009 года работаю только с Сетью (веб) и консольными приложениями. Хотя основная моя задача заключается в создании веб-сайтов, мне очень часто приходится писать библиотеки кода C# или консольные приложения, которые выполняют какие-либо дополнительные действия.

На консольных приложениях мы уже практикуемся большую часть книги, и настало время познакомиться с веб-программированием.

Все больше различных приложений переходят в Интернет. Если в 1990-х годах корпоративные программы писали в виде десктопных приложений и чаще всего для Windows, то уже в 2010-х начали все больше писать корпоративных веб-приложений, которые можно запускать из-под любой ОС.

Последние несколько лет я работаю для рынка США и Канады над приложениями по расчетам зарплаты, ведению кадрового учета и всего, что с этим связано. И если раньше подобное создавали для Windows, то сейчас — для браузера, чтобы можно было работать из Windows, Linux или macOS.

13.1. Создание первого веб-приложения

До сих пор мы создавали консольные приложения, но и создание приложений для Сети от этого отличается несильно. Мы также должны в среде Visual Studio выбрать создание нового проекта и в выпадающем списке языков все также выбрать C#, но вот в списке типа проектов выбрать уже **Web**.

Мы начнем создание веб-проектов с самого начала — с пустого проекта, поэтому из списка предлагаемых шаблонов выберите **ASP.NET Core Empty** (рис. 13.1). Да, при создании проекта мы выбираем шаблон, но после его создания можем добавить в код то, что нам понадобится, или убрать то, что окажется лишним. Мы будем строить MVC-приложение (о приложениях MVC рассказано чуть далее), поэтому, чтобы в будущем в своих проектах не делать базовые вещи вручную, вы можете выбрать также шаблон **ASP.NET Core Web App (Model-View-Controller)**.

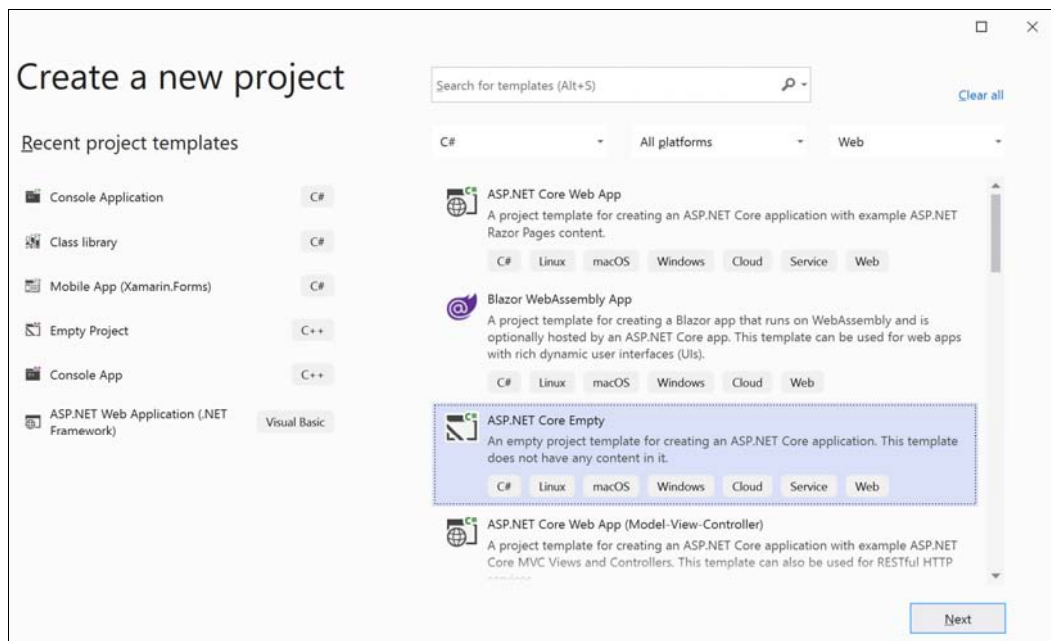


Рис. 13.1. Создаем пустой веб-проект

Второй шаг будет практически таким же, как и при создании консольных приложений, — мы должны задать имя проекта, имя решения и папку, где будет располагаться проект. А вот на третьем шаге появятся новые параметры, и, помимо выбора .NET платформы, мы можем здесь выбрать:

- ❑ **Configure for HTTPS** — сконфигурировать для работы с шифрованием трафика HTTPS. Все современные сайты рекомендуется делать с шифрованием, потому что это безопаснее, но далеко не каждый хостинг поддерживает бесплатные сертификаты, которые необходимы для поддержки шифрования трафика, а платные сертификаты могут стоить очень дорого. Подключить или отключить HTTPS можно в любой момент, и я оставлю этот параметр по умолчанию включенным;
- ❑ **Enable Docker** — включить поддержку контейнеров, что очень популярно с точки зрения публикации веб-приложений .NET. Эту поддержку тоже можно включить в любой момент, но пока дополнительно включать ее не станем;
- ❑ **Do not use top-level statements** — не использовать операторы верхнего уровня. Этот параметр был и у консольных приложений, и если его не выбрать (поведение по умолчанию), то в файле Program.cs не будет метода Main.

Итак, мы создали приложение, и давайте теперь посмотрим на окно **Solution Explorer** (Проводник решения), показанное на рис. 13.2. В корне дерева находится имя решения — в моем случае я дал ему название **MyWebSite**. Сразу под ним идет имя проекта — **MyWebSite**. Пока всё так же, как у консольных приложений.

Самые интересные опции находятся именно в проекте, и тут много нового:

- ❑ **Dependencies** — папка зависимостей. Здесь будут расположены библиотеки, которые необходимы для работы приложения. Устанавливая новые пакеты, мы

можем расширять возможности, которые будут ему доступны. Если сайту, например, нужна возможность работы с ZIP-архивами, то мы можем установить в папку зависимостей необходимую для этого функциональность. До сих пор нам не приходилось сталкиваться с пакетами, но, имея дело с веб-программированием, нам скоро понадобится расширять свои возможности;

- у пустого проекта в папке **Dependencies | Framework** (Фреймворк) содержатся два пакета: **Microsoft.AspNetCore.App** и **Microsoft.NetCore.App**. Первый из них предоставляет базовые возможности веб-приложений платформы .NET, а второй — базовые возможности самого .NET, и мы до сих пор в основном использовали возможности именно второго;
- **Properties** — папка свойств. Здесь находится файл настроек `launchSettings.json`.

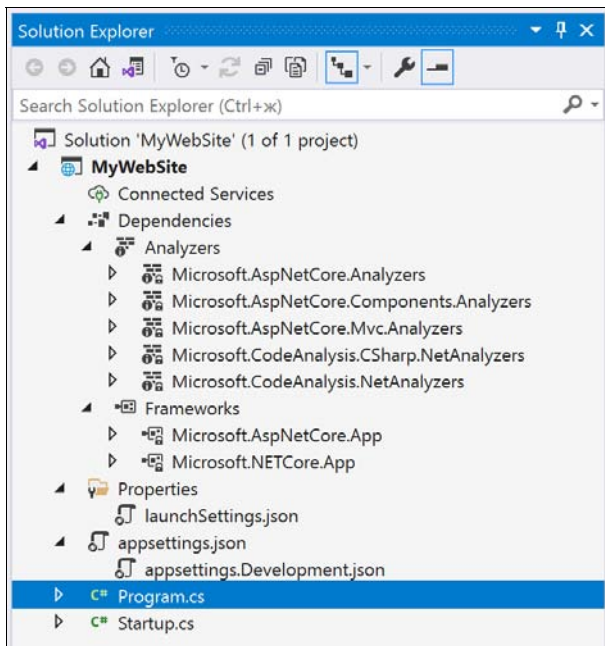


Рис. 13.2. Окно Solution Explorer

Помимо папок, в состав проекта входят также файлы: `Program.cs` — в нем содержится метод `Main`, с которого начинается выполнение приложения, и `Startup.cs`, содержащий код, который конфигурирует сайт.

Еще раз отмечу, что если вы создали проект с использованием операторов верхнего уровня (не установили флажок **Do not use top-level statements**), то, как и в случае с консольными приложениями, метода `Main` в файле `Program.cs` вы не увидите, а вместо него там сразу же будет прописан код. При этом файл `Startup.cs` тоже будет отсутствовать. Весь код конфигурации в этом случае нужно будет писать прямо в файл `Program.cs`.

И т. к. операторы верхнего уровня становятся поведением по умолчанию, то далее я буду рассматривать именно этот шаблон. Если же вы когда-либо столкнетесь с веб-

приложением, где в файле `Program.cs` присутствует метод `Main`, то его конфигурацию ищите в файле `Startup.cs`.

Запустите проект (клавишей <F5>), чтобы увидеть результат. Консольные приложения мы начинали изучать с приветствия «Hello, World!» на черном экране консольного окна, вот и веб-приложение встречает нас таким же приветствием в окне браузера (рис. 13.3).

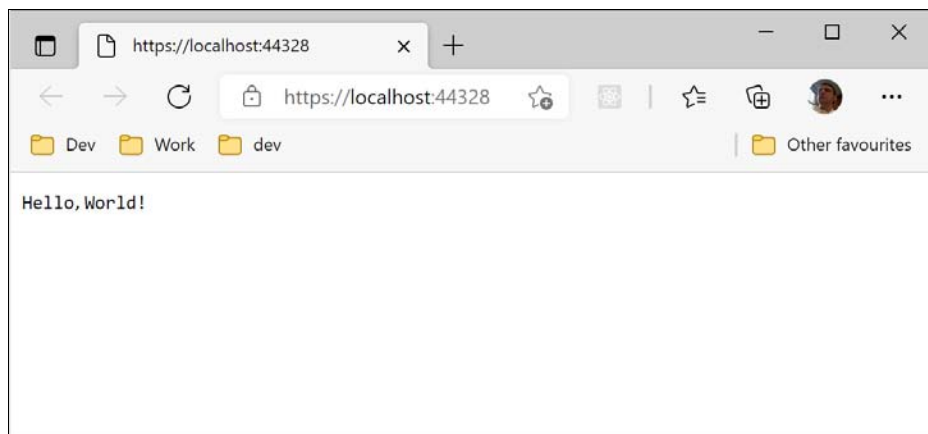


Рис. 13.3. Приветствие от веб-приложения

В адресной строке интернет-адрес (URL) приложения начинается с **https://**, потому что я оставил настройку шифрования трафика по умолчанию. После этого в адресе следует **localhost**, что всегда соответствует локальному компьютеру. Порт **44328** — это конфигурируемый параметр, и вы можете его менять, если захотите, потом в настройках приложения.

До сих пор мы работали с консольными приложениями, включающими один-два файла с исходными кодами. С погружением в веб-программирование количество файлов в нашем проекте будет значительно увеличиваться, поэтому настало время чуть ближе познакомиться с файлом проекта.

В .NET Core/.NET 7 немного поменяли политику работы проектов с файлами по сравнению с .NET Framework. В .NET Framework мы должны были явно добавлять файлы в проект на Visual Studio, даже если они находились в одной из его папок. Теперь если папка принадлежит проекту, то все файлы по умолчанию тоже добавляются в него.

Помню, когда я работал на Sony, то автоматическое добавление файлов в проект мы реализовывали самостоятельно через сторонние утилиты для сборки проектов. Похоже, MS решила пойти тем же путем, и это правильно.

Попробуйте скопировать в файловой системе какой-нибудь файл в папку, где находится проект, и он автоматически появится в составе проекта Visual Studio. Это нереально удобно, и я рад, что MS наконец сделали это. Стоит удалить файл в файловой системе, и он исчезнет и из проекта.

Выделите имя проекта в окне **Solution Explorer** — вы должны увидеть содержимое файла проекта, и здесь же вы можете менять настройки. Если содержимое файла не отобразилось, то щелкните правой кнопкой на имени проекта и из контекстного меню выберите **Edit Project File** (Редактировать Файл Проекта). То же самое можно увидеть, если зайти в папку с проектом и открыть файл `MyWebSite.csproj` в любом текстовом редакторе.

Содержимое файла очень простое:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
</Project>
```

Здесь у тега `Project` мы указываем SDK для работы с Сетью: `Microsoft.NET.Sdk.Web`. В атрибуте `TargetFramework` указывается фреймворк, который мы хотим использовать, и в моем случае это `.NET 7` — т. е. как раз тот параметр, который я выбрал при создании проекта.

Если у вас уже есть какой-либо проект, который использует `.NET 5`, переход на новую версию сводится к простому изменению версии в этом теге на более новую. После этого во время компиляции могут возникнуть проблемы, но в последнее время Майкрософт не так часто вводит изменения, которые ломают код.

В файле проекта могут также содержаться указания на файлы или папки — например, следующий код явно подключает к проекту определенную папку:

```
<ItemGroup>
  <Folder Include="Sources/" />
</ItemGroup>
```

Если у вас какой-то файл или папка не появляются в проекте или компилятор не видит код в файле, стоит заглянуть в файл проекта и посмотреть, нет ли там каких-либо аномалий, которые вы не добавляли сами. Например, следующие строки в файле проекта приведут к тому, что компилятор просто не будет видеть код в файле `Program.cs`:

```
<ItemGroup>
  <Compile Remove="Program.cs" />
</ItemGroup>
```

Здесь в теге `Compile` (компилятор) содержится атрибут `Remove` (убрать) и указан файл `Program.cs` — естественно, что этот файл не будет компилироваться, хотя и останется принадлежащим проекту. У меня уже было несколько случаев, когда среда разработки не видела код или файлы в папке с проектом. Если вы сами не добавляли в файл проекта что-либо вроде тега `Compile Remove`, то просто уберите из него эту строку.

Переходим к рассмотрению CS-файлов, которые остались в корне проекта, — это наш Program.cs:

```
var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", () => "Hello, World!");

app.Run();
```

Еще раз напомним: если ваш файл выглядит иначе, то вы не используете операторы верхнего уровня, — тогда ищите этот код в файле Startup.cs.

Первая строка создает `WebApplicationBuilder`, через который мы будем включать необходимую нам функциональность. Во второй строке строится приложение, и мы получаем в результате экземпляр `WebApplication`, через который будем конфигурировать веб-приложение.

Ну и далее следует запуск метода, который будет выполняться в ответ на любой входящий запрос:

```
app.MapGet("/", () => "Hello, World!");
```

За время существования .NET эта строка менялась несколько раз, и сейчас она выглядит наиболее удачно. Так что если вы увидите где-либо немного другой код, то не пугайтесь — это просто старый подход.

Самая последняя строка: `app.Run` — уже непосредственно запускает приложение.

Версия `MapGet`, которую мы видели в третьей строке, — это минимальная версия с двумя параметрами. Первый ее параметр — это интернет-адрес (URL), для которого мы хотим написать код. В нашем случае — это просто слеш: `/`, что соответствует главной странице. Второй параметр — это анонимная функция, которая просто возвращает строку, которую мы и увидим в браузере.

Мы можем вызывать `MapGet` несколько раз, но в первом параметре каждый раз должна быть указана разная строка адреса. Если указать дважды одну и ту же, то сервер не будет знать, какую из них выполнять.

Например, следующая строка добавляет код, который будет выполняться при обращении к адресу `/test`:

```
app.MapGet("/test", () => "Это тест");
```

Эту строку нужно написать до `app.Run()`, которая должна быть последней в файле. При отображении русского языка у браузера могут возникнуть проблемы, потому что он не знает, какую кодировку использовать. Если что-то в этом смысле пойдет не так, то используйте пока английский. Когда же мы будем работать с HTML, то сможем использовать универсальную кодировку UTF-8.

Запустите сайт и в строке адреса укажите `https://localhost:7261/test` — вы должны увидеть свою новую строку.

В качестве второго параметра `MapGet` мы можем использовать асинхронную функцию, которая будет получать в качестве параметра контекст типа `HttpContext`, через

который можно получить доступ к информации о том, с какого адреса пришел запрос, какие параметры были переданы в качестве URL, какие данные сохранены в Cookie пользователя, и многое другое.

В следующем примере я создаю маппинг для адреса `/param`:

```
app.MapGet("/param", async context =>
{
    await context.Response.WriteAsync("Это строка параметров:"
        + context.Request.QueryString.ToString());
});
```

Метод `MapGet` получает здесь два параметра: первый параметр — строка, в которой содержится URL, а второй — метод, получающий в качестве параметра контекст отображения страницы (типа `HttpContext`), в который можно писать текст для вывода на страницу. Так что все, что находится в круглых скобках во втором параметре (после запятой), — это на самом деле анонимная (без имени) функция и ее код.

В приведенном примере я уже не просто отображаю сообщение, но еще и содержимое строки запроса из браузера: `context.Request.QueryString.ToString()`. Запустите сайт, в строке адреса укажите `/param?test`, и в результате вы этот запрос должны увидеть на странице.

То же самое можно переписать с использованием именованной функции:

```
app.MapGet("/func", HomePage);
```

Здесь мы передаем URL и имя метода: `HomePage`, но нам нужно также объявить в конце файла этот метод `HomePage`:

```
async Task HomePage(HttpContext context) {
    await context.Response.WriteAsync("Hello, World!");
}
```

Теперь функция с именем `HomePage` получает параметр `context`, и здесь мы видим тип этой переменной: `HttpContext`. В результате просто выводится сообщение.

Приведенный код говорит, что нужно записать в результат `Response` строку. Контекст `context` — это специальная переменная (типа `HttpContext`), через которую мы можем получить информацию о запросе, который направил пользователь, и через свойство `Response` записать ответ, который будет отправлен пользователю.

До сих пор наш код на запросы браузера возвращал просто тест. Чтобы запросы возвращали HTML-код, мы должны указать тип контента `text/html` в поле `context.Response.ContentType`:

```
context.Response.ContentType = "text/html";
```

Это и показано в следующем примере:

```
app.MapGet("/html", async context =>
{
    context.Response.ContentType = "text/html";
```

```
await context.Response.WriteAsync("<h1>Привет,</h1> Мир!"  
    + context.Request.QueryString.ToString());  
});
```

Вот, пожалуй, самый минимальный код, который можно использовать для запуска собственного сайта.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter13\MyWebSite* сопровождающего книгу электронного архива (см. приложение).

13.2. Работа с конфигурацией сайта

Если у вас уже есть опыт создания веб-приложений на .NET, то вы, скорее всего, работали с конфигурационными файлами в XML-формате. Самый основной конфигурационный файл *web.config* тоже имеет XML-структуру.

Сейчас разработчики .NET решили перейти на формат JSON, который более легкий, занимает меньше места на диске, и на нем проще писать парсер (код для чтения), а значит, он и работать будет быстрее.

Начнем рассмотрение конфигурации с файла *launchSettings.json* из папки *Properties*. В самом начале файла вы можете увидеть следующую конфигурацию для IIS:

```
"iisSettings": {  
  "windowsAuthentication": false,  
  "anonymousAuthentication": true,  
  "iisExpress": {  
    "applicationUrl": "http://localhost:51684/",  
    "sslPort": 44332  
  }  
},
```

После запуска приложения — когда мы загружали сайт по HTTPS (а это поведение по умолчанию) — то видели после имени *localhost* номер порта 44328 (см. рис. 13.3). Вот как раз в этом файле и настраивается номер порта, и вы можете его без проблем поменять на любой другой, если у вас сайт не запускается из-за того, что указанный порт уже занят кем-то другим. Работая с несколькими сайтами на одном компьютере, во избежание конфликтов приходится задавать разные порты.

Чуть ниже в этом файле прописаны профили. Среди профилей могут быть сгенерированы три секции: *http*, *https* и *IIS Express*. В зависимости от ситуации может использоваться тот или иной профиль. У профиля *https* есть адреса, среди которых присутствует и номер порта, через который будет осуществляться доступ при запуске из командной строки:

```
"https": {  
  "commandName": "Project",  
  "dotnetRunMessages": true,
```

```
"launchBrowser": true,
"applicationUrl": "https://localhost:7261;http://localhost:5099",
"environmentVariables": {
  "ASPNETCORE_ENVIRONMENT": "Development"
}
},
```

Самое важное здесь — это переменные окружения `environmentVariables`, где для переменной `ASPNETCORE_ENVIRONMENT` указано `Development`.

При настройке приложения бывает необходимо создавать различные параметры в зависимости от того, находимся мы сейчас в режиме разработки или выполняем сайт на рабочих серверах.

Так, если мы работаем в режиме разработки локально, то можем смело включить отображение ошибок прямо на странице. На рабочих серверах ошибки лучше на странице не показывать, потому что это выглядит некрасиво и может дать хакеру важную информацию.

В следующем примере — для работы в режиме разработки — отображение ошибок разработчика включается с помощью параметра `UseDeveloperExceptionPage`:

```
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
```

Эта проверка как раз вернет истину, если переменная `ASPNETCORE_ENVIRONMENT` равна `Development`.

Теперь перейдем к конфигурации приложения. При создании нового проекта среда Visual Studio уже создала для нас конфигурационный файл по умолчанию: `appsettings.json`. У меня в этом файле сразу имеется несколько строк:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

Эту информацию будет использовать веб-сервер Kestrel, который и станет выполнять код, когда мы запускаем приложение из Visual Studio или командной строки. Тут ему сообщается, что нужно по умолчанию использовать уровень регистрации события `Warning` и разрешать подключения со всех хостов. Это всё.

Опять же, если вы работали со старым ASP.NET, то, скорее всего, видели его файл `web.config`, содержащий огромное количество настроек. В нашем случае конфигу-

рационный файл `appsettings.json` содержит два параметра, да и то необязательные. Попробуйте удалить из этого файла все строки и оставить только:

```
{
}
```

Это необходимо, чтобы JSON-файл оставался корректным. Запустите сайт, и он выполнится.

Попробуйте теперь удалить даже эти две строки — т. е. оставить файл совершенно пустым, и запустить сайт. На этот раз произойдет исключительная ситуация (рис. 13.4).

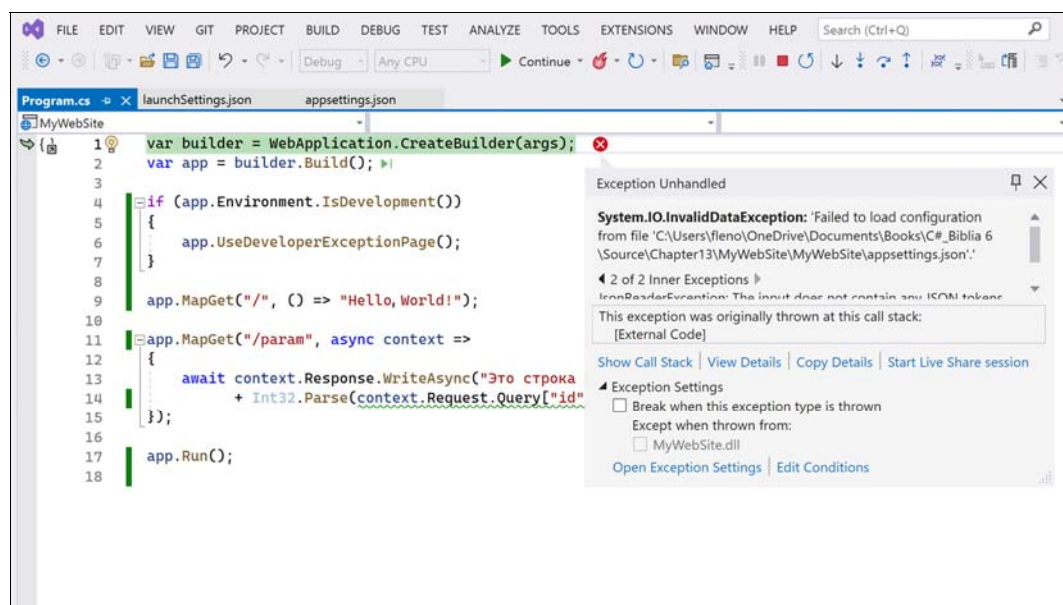


Рис. 13.4. Ошибка из-за испорченного файла конфигурации

Глядя на место, где произошла ошибка, можно догадаться, что конфигурация из файла `appsettings.json` загружается при создании веб-хоста в методе `WebApplication.CreateBuilder`.

А что если этот файл даже не создан или мы его удалим? Сайт будет работать. Kestrel сможет запустить на выполнение сайт, если его конфигурационный файл корректный либо отсутствует. Можете попробовать удалить в файловом менеджере файл `appsettings.json` и запустить сайт — и он выполнится.

Давайте добавим в конфигурационный файл еще один параметр: `HelloMessage` — и именно его значение будем отображать на странице:

```
{
  . . .
  . . .
  "AllowedHosts": "*",
```

```
"HelloMessage": "Привет из .NET"  
}
```

В файле `Program.cs` вы можете получить доступ к конфигурации через `app.Configuration`. Это свойство — класс, который реализует интерфейс `Configuration`, и у него есть индексатор, через который можно получить по ключу любое значение из конфигурации.

Мы создали параметр с именем `HelloMessage`, а значит, в качестве ключа нужно передать именно его:

```
app.MapGet("/", () => app.Configuration["HelloMessage"]);
```

Если сейчас запустить приложение, то вы должны увидеть наше сообщение на странице.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\WebsiteConfig` сопровождающего книгу электронного архива (см. *приложение*).

13.3. Работа со статичными файлами

В проекте есть одна очень интересная папка — `wwwroot`. В ней нужно располагать файлы, которые должны быть доступны при обращении к сайту. У нас в шаблоне пустого веб-приложения этой папки нет, но ее можно создать — главное, дать ей имя `wwwroot`.

Создать эту папку можно двумя способами: обычным путем в файловой системе или щелкнуть правой кнопкой мыши на имени проекта, выбрать **Add | New Folder** и здесь указать имя: `wwwroot`. Обратите внимание, что значок папки сразу же изменится на глобус. Если вы создавали эту папку из файловой системы, то она может не появиться в Visual Studio, потому что в ней ничего нет. Но стоит только скопировать туда хотя бы один файл, она сразу же появится в Проводнике решения.

Если сейчас заглянуть в эту папку, то там могут находиться какие-то файлы — это зависит от шаблона, который использовала Visual Studio. Под Windows в пустом шаблоне папка пустая, а под macOS у меня там оказались файлы. Шаблон **Empty** под macOS вообще содержит намного больше файлов и функций.

Если у вас в папке `wwwroot` нет файлов, попробуйте сейчас добавить в нее какой-нибудь JPEG-файл, запустить сайт и обратиться к нему из браузера. В примере для этого раздела, приведенном в сопровождающем книгу электронном архиве, в папке `wwwroot` находится файл `alberta.jpg`, но если после запуска сайта попытаться перейти по адресу: <https://localhost:7261/alberta.jpg>, то произойдет ошибка.

Дело в том, что в ASP.NET все возможности конфигурируются, в том числе и доступ к статичным файлам. В .NET Framework это легко можно было сделать через файл `web.config`, а в .NET Core/.NET 7 следует использовать специальный метод в `IApplicationBuilder`. Так что открываем файл `Program.cs` и где-то после создания приложения и получения переменной `app` добавляем одну строку:

```
app.UseStaticFiles();
```

Всего одна строка кода, но теперь после компиляции и запуска сайта браузер сможет получать с сервера файлы, в том числе и изображения. Так что при обращении к <https://localhost:7261/alberta.jpg> изображение загрузится.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\StaticFiles` сопровождающего книгу электронного архива (см. приложение).

13.4. Модель – Представление – Контроллер

Мы увидели, как можно создать базовое приложение, но построение логики в одном-единственном файле для большого сайта может стать серьезной «занозой», да и как поддерживать такое, я даже не представляю. Можно попытаться создать какую-то машину, которая будет смотреть на то, какие параметры присылает браузер серверу, и попробовать в зависимости от различных составляющих вызывать тот или иной код, который мы для своего удобства разбрасываем по файлам.

Казалось бы, отличный план, но совершенно не нужный, потому что в .NET все это уже реализовано. Microsoft неплохо потрудились, чтобы предоставить нам все необходимое для возможности создавать приложения с использованием шаблона MVC (Model-View-Controller — Модель-Представление-Контроллер), основанного на разделении логики приложения и отображения его страниц.

MVC, наверное, самый популярный шаблон в веб-программировании, и его смысл заключается в том, чтобы отделить бизнес-логику приложения (Model) от представления (View). Контроллер (Controller) — это промежуточное звено, которое связывает бизнес-логику и то, как отображается страница.

С ростом популярности мобильных приложений этот шаблон показал свои лучшие стороны, если он реализован правильно. Мы сможем использовать для модели один и тот же код, потому что модель может реализовываться для двух разных приложений, где представлениями могут служить: веб-интерфейс на основе HTML или пользовательский интерфейс (UI) мобильного приложения, а связывать их будут свои контроллеры.

Чтобы не запутаться в контроллерах, они должны быть максимально «глупыми». Их основная задача — получить данные от модели и передать представлению. Когда же представление хочет отправить данные модели, то контроллер получает информацию от представления, может конвертировать данные в нужный формат и передать дальше модели.

Возможный вариант использования одной и той же логики для разного типа приложений показан на рис. 13.5. Мобильные контроллеры могут быть организованы в виде кода, который — если использовать фреймворк Xamarin — передает данные напрямую UI, а может присутствовать и еще один веб-контроллер, который отображает данные в виде JSON, и это уже получает сервис WebAPI REST.

За счет такого разделения представление можно без проблем менять или даже иметь сразу несколько его вариантов. Одна и та же модель может работать как для веб-сайтов, так и для мобильных приложений.

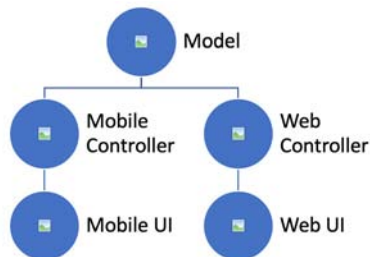


Рис. 13.5. Возможный вариант использования одной и той же логики для разного типа приложений

Реализация MVC в .NET Core/.NET 7 ничем не отличается по своей идеологии от других языков и фреймворков, но пару слов все же скажу на всякий случай.

Когда вы обращаетесь к определенному URL, то первым делом специальный код (маршрутизатор) определяет, какой контроллер должен быть вызван. И далее уже контроллер берет на себя управление. Он по мере необходимости может создавать один или более объектов-моделей и отвечает за то, какое представление отобразить пользователю.

Модель — это простые классы C#, которые отличаются от других классов только логически. Просто на них возлагается ответственность за реализацию «мозга» нашего приложения. Конечно же, эту логику желательно как-то группировать, а как это делать, зависит уже от проекта и предпочтений. В простых проектах как минимум создают отдельную папку Model, куда помещают всю логику. Кто-то предпочитает отделять модели более кардинально — создавать отдельные библиотеки и хранить логику в DLL-файлах. Ну а в больших проектах отдельные DLL-файлы становятся просто необходимостью.

Контроллеры — это тоже классы C#, но они происходят от класса `Controller`:

```

public class HomeController: Controller
{
    public IActionResult Index()
    {
        // Здесь может быть код создания моделей

        return View();
    }
}
  
```

Здесь создается класс `HomeController`, который происходит от базового контроллера `Controller`. В приведенном примере есть только один метод: `Index`, но их может быть и больше. По умолчанию подобные методы будут вызываться в ответ на пользовательский запрос `/home/index`, но об этом чуть позже, когда мы начнем говорить про маршрутизацию запросов. В качестве результата такой метод возвращает `View`.

Отделение логики от представления позволяет не только облегчить замену дизайна, но и бизнес-логики. Я много раз сталкивался с задачами, когда нужно было менять

или улучшать внешний вид страниц. За счет того, что при этом приходилось менять только HTML-файлы, определяющие внешний вид, без необходимости трогать логику, все проходило очень даже гладко.

13.5. Маршрутизация

Если сайт состоит из одной страницы, то можно все реализовать прямо в файле `Program.cs`, но, как мы уже увидели, даже для одного-единственного файла используется *маршрутизация*. А если сайт больше одной страницы, то тут лучше структурировать код несколько иначе и задействовать всю мощь MVC. С теорией вопроса мы уже познакомились в предыдущем разделе, а теперь пора все увидеть на практике.

Итак, для начала нужно включить MVC, потому что по умолчанию в приложениях .NET ничего не включено, и нужно это делать самостоятельно.

Включать подобные возможности надо в файле `Program.cs`:

```
builder.Services.AddControllersWithViews();
builder.Services.AddMvc();
```

Здесь мы подключаем контроллеры вызовом методов `AddControllersWithViews()` и `AddMvc()`. При этом если нужно выполнить какой-то метод, который начинается с `builder.Services`, то его нужно записать до вызова `builder.Build()`. То есть указанные строки должны быть записаны ранее строки:

```
var app = builder.Build();
```

Теперь нужно включить маршруты:

```
app.UseRouting();
```

Так как мы тут обращаемся к `app`, то это уже нужно делать после вызова `builder.Build()`.

MVC — это уже не просто небольшая функциональность, поэтому приходится сделать несколько шагов заранее, чтобы потом в книге можно было их использовать. Как минимум нужны контроллеры, которые мы уже подключили, нужны маршруты, которые мы включили, остается только настроить маршруты.

Итак, последний этап — конфигурация маршрутов, чтобы они направляли запросы на контроллеры, которые как раз и управляют всем «оркестром» (т. е. приложением).

Убираем из своего кода вызов метода `UseEndpoints` со всем кодом, который мы в нем писали. Самый простой способ настройки маршрутов — использовать версию по умолчанию, которой хватит для большинства проектов. Для этого можно вызвать следующую строку:

```
app.MapDefaultControllerRoute();
```

Но я не буду использовать этот метод, а настрою то же самое вручную, чтобы показать, как это реально можно сделать. Раньше для настройки маршрутов нужно было выполнить такой код:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

Сейчас он немного упростился, потому что у `app` есть метод `MapControllerRoute` и его нужно вызывать напрямую:

```
app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}"
);
```

Здесь вместо конфигурирования отдельных URL с помощью `MapGet` я включаю маршрутизацию на контроллеры с помощью метода `MapControllerRoute`. Этому методу нужно передать как минимум два параметра: имя и шаблон. Я не стал изобретать здесь велосипед, а использую классический пример: в роли имени выступает просто `default` (по умолчанию), а в качестве шаблона:

```
{controller=Home}/{action=Index}/{id?}
```

Такой же шаблон будет установлен, если просто выполнить:

```
MapDefaultControllerRoute
```

В фигурных скобках указываются разделенные слешами части шаблона, которых в нашем случае три. Рассмотрим каждую часть:

- ❑ `controller=Home` — указывает, что здесь первым должно идти имя класса контроллера. После знака равенства стоит имя по умолчанию на тот случай, если контроллер не указан. Это значит, что если пользователь пытается просто загрузить сайт `www.flenov.info` без указания слеша в конце и имени контроллера, то будет выполнен контроллер `HomeController`. Да, класс контроллера рекомендуется указывать со словом `Controller` в конце. Если же пользователь введет в браузере `www.flenov.info/blog`, то маршрутизатор MVC будет искать класс `BlogController`, потому что первым словом в адресе после слеша идет `blog`;
- ❑ `action=Index` — параметр `action` позволяет указать, какой метод класса контроллера нужно вызвать для обработки запроса. После знака равенства снова указывается имя метода по умолчанию. Если пользователь обращается к сайту `www.flenov.info/blog`, то маршрутизатор MVC будет искать класс `BlogController` и метод `index` в нем, потому что явно имя метода не указано. Если же ввести `www.flenov.info/blog/list`, то это будет соответствовать контроллеру `BlogController` и методу `List`:

```
public class BlogController : Controller
{
    public IActionResult Index()
```

```
{
    return View();
}
public IActionResult List()
{
    return View();
}
}
```

Это самая простая реализация класса `BlogController` и двух методов: `Index` и `List`.

Согласно нашему шаблону первое слово в адресе после слеша — это имя контроллера, второе слово — это имя метода, и у каждой из этих составляющих есть значение по умолчанию;

- `{id?}` — последняя часть нашего шаблона, параметр метода. Он не обязательный, и об этом говорит знак вопроса. То есть последняя часть может отсутствовать вовсе, и никаких ее значений по умолчанию не будет. Если же пользователь попытается загрузить URL с тремя частями, то первая будет контроллером, вторая — методом контроллера, а третья — параметром. Так что `Id` в нашем случае — это имя параметра.

Допустим, пользователь загружает адрес `www.flenov.info/blog/show/10`. Тогда маршрутизатор вызовет метод `show` у контроллера `BlogController` и передаст 10 через параметр `id`:

```
public class BlogController : Controller
{
    public string show(string id)
    {
        return "ID Value = " + id;
    }
}
```

Итак, в теории мы с маршрутом познакомились, теперь посмотрим, как это реализуется на практике. У нас приложение минимальное, поэтому нужно создать отдельную папку для контроллеров и отдельную — для представлений. Щелкаем правой кнопкой мыши на имени проекта, выбираем из контекстного меню **Add** и потом **Folder**, чтобы создать папку, и даем ей имя `Controllers`. Имя папки очень важно и должно быть написано именно так.

В папке `Controllers` создаем файл `HomeController.cs`, для чего щелкаем правой кнопкой мыши на папке `Controllers` и выбираем **Add | New File**. В открывшемся диалоговом окне слева выбираем **ASP.NET Core**, справа — **MVC Controller** и внизу вводим имя файла. Для нашего примера пусть это будет `HomeController.cs`.

При создании нового файла вы можете выбирать в этом диалоговом окне любой шаблон, не обязательно всегда искать **MVC Controller**. Выбранный шаблон всего лишь говорит о том, что надо поместить в файл после создания. При создании контроллера можно выбрать и шаблон **Class**, а потом добавить нужное содержимое вручную.

Давайте поместим в только что созданный файл контроллера следующий код:

```
using Microsoft.AspNetCore.Mvc;

namespace MyWebSite
{
    public class HomeController : Controller
    {
        public string Index(string id)
        {
            return "ID Value = " + id;
        }
        public string list(string id)
        {
            return "return list";
        }
    }
}
```

Представлений у нас пока нет, и контроллер возвращает лишь строку, а в этом случае MVC просто отобразит эту строку на странице. Запустите сайт нажатием клавиши <F5>, и на странице должна отобразиться надпись: **ID Value =**.

Значения нет, потому что мы ничего не передали. Когда браузер запустится, то адрес страницы будет, скорее всего: <https://localhost:7261>. Никакого пути нет, поэтому все загрузится по умолчанию. Это то же самое, что загрузить <https://localhost:7261/home/index>. Здесь в URL указано `home/index`, и согласно нашему маршруту это значения по умолчанию для первой и второй частей адреса. Попробуйте передать через адрес <https://localhost:5001/home/index/100> еще и параметр, и на странице вы должны будете увидеть число **100**.

У нашего контроллера `HomeController` есть также метод `list`, и чтобы вызвать его в браузере, введите: <https://localhost:5001/home/list>.

Это, наверное, самый популярный способ работы с маршрутизацией URL, но не единственный.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter13\ Controllers* сопровождающего книгу электронного архива (см. приложение).

Настройка маршрутов с помощью `MapControllerRoute` в методе `Configure` класса `Startup` является очень даже гибким и эффективным методом, потому что позволяет реализовать классический подход, когда после имени домена идут два компонента, из которых состоит действие, которое хочет выполнить пользователь. Среди популярных действий можно выделить `blog/list`, `blog/show/{id}`, `catalog/browse`, `cart/show`, `checkout/shipping`, `checkout/payment`. Это все очень удобно, но бывают случаи, когда необходимо решить более сложные варианты маршрутов и организовать их с помощью шаблона может быть не так уж легко.

На мой взгляд, более гибким решением нестандартных маршрутов является использование *атрибутов*. Мы можем все маршруты делать через шаблоны или все маршруты делать через атрибуты, а можем использовать и то и другое одновременно. Я предпочитаю как раз третий подход, когда базовые случаи покрываются шаблоном, а нестандартные ситуации решаются атрибутами.

Давайте создадим новый контроллер. Откройте проект Visual Studio, щелкните правой кнопкой мыши на папке **Controllers** и выберите создание нового файла. В открывшемся окне выберите **MVC Controller | Class**, или можно выбрать даже любой другой тип CS-файла, ведь главное — это расширение и имя, а имя для нашего теста пусть будет `TestController`.

Начнем с простого варианта:

```
using Microsoft.AspNetCore.Mvc;

namespace MyWebSite.Controllers
{
    public class TestController : Controller
    {
        public string Index()
        {
            return "Test Index method";
        }
    }
}
```

Пока ничего нового — просто класс `TestController`, который происходит от базового контроллера `Controller`. Метод по умолчанию здесь: `Index`. С маршрутизацией по умолчанию, которую мы рассматривали чуть ранее, обращение к этому методу через браузер будет таким: `/test/index`. А что если мы хотим сделать, чтобы URL выглядел так: `/my/test/index`?

Проблему можно решить двумя способами: первый (старый) — добавить еще один маршрут. Открываем файл `Startup.cs` (если его нет, то `Program.cs`), находим в `Configure` вызов метода `UseMvc` и добавляем туда еще один шаблон маршрута:

```
app.MapControllerRoute(
    name: "Test",
    pattern: "my/{controller=Home}/{action=Index}/{id?}");

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Здесь я дважды вызываю `MapControllerRoute`. Первый параметр `Name` должен быть уникальным и просто отражать смысл маршрута. Сначала желательно вызывать как можно более ограниченный маршрут, а потом уже более общий. Вариант по умолчанию относится как раз к общим. Наш новый — более ограниченный, потому что должен работать только тогда, когда URL начинается с `my/`.

Запускаем приложение и можем убедиться, что все работает, и теперь при обращении к `/home/index` и `/my/test/index` открываются две уникальные страницы.

Да, старый способ работает, но есть более удобный вариант — с использованием атрибутов.

Перед именем класса и метода можно написать атрибут `Route` и в нем перезаписать часть URL.

В следующем примере я показал пару интересных примеров того, как можно через атрибуты указать маршрут:

```
using Microsoft.AspNetCore.Mvc;

namespace MyWebSite
{
    [Route("my/test")]
    public class TestController : Controller
    {
        [Route("show")]
        public string Index()
        {
            return "Index Test method";
        }

        [Route("details/{id}")]
        public string Details(string id)
        {
            return "ID Value = " + id;
        }
    }
}
```

Здесь перед строкой объявления класса я указываю, что все методы класса будут вызываться только тогда, когда URL начинается с `my/test`. В случае с классом в качестве атрибута можно использовать `Route` (раньше еще поддерживался атрибут `RoutePrefix`, который убрали, и неизвестно, вернут или нет).

Перед именами методов можно указывать оставшуюся часть URL. Для метода `Index` я указал `show`, а это значит, что если загрузить URL `/my/test/show`, то будет вызван метод `Index` класса `TestController`.

Для метода `Details` указано `details/{id}`. Здесь фигурные скобки вокруг `id` не случайны. Этим мы указываем, что значение в этом месте будет передано самому методу в качестве параметра `id`. Так что, если обратиться по адресу: `/my/test/details/4`, то будет вызван метод `Details` класса `TestController`, а число 4 как раз находится там, где у нас шаблон `{id}`, — это число и передадут в качестве параметра `id`.

С помощью атрибутов мы можем привязывать код не только к URL, но и к методам доступа. Есть несколько различных методов доступа, основным из которых являет-

ся GET. Именно этот метод используется, когда мы в браузере вводим какой-то адрес и просим сервер вернуть нам контент для этого адреса. В этом случае параметры адреса передаются в URL, и мы их видим после вопросительного знака ? в виде списка `имя=значение`, разделенного символом `&`.

Второй по популярности метод — POST, это основной метод отправки данных на сервер. Например, когда вы вводите данные в форме регистрации или в форме входа на сайт, то данные отправляются методом POST, и в этом случае введенные данные будут находиться в теле запроса, а не в строке URL.

Для нас сейчас важно понимать, что мы можем привязать разный код к каждому из методов, и это действительно необходимо. Например, если вы вводите в строке URL адрес для входа на мой сайт: `https://www.flenov.info/login/index`, то на сервер будет направлен GET-запрос. Сервер просто должен вернуть содержимое страницы. Когда же вы введете данные в форму и нажмете кнопку **Войти**, то на тот же URL будет направлен запрос POST, и на сервере станет выполняться уже другой код, который будет проверять значения в форме, а если введен правильный e-mail, то он еще и проверит, действительно ли этот пользователь существует в базе данных, и правильные ли данные введены.

Возможная привязка кода не только по URL, но и по методу доступа позволяет использовать в разных ситуациях разный код.

Чтобы наш код выполнялся на определенный метод запроса, перед именем метода надо поставить атрибут `[HttpGet]` или `[HttpPost]`:

```
[HttpGet]
public string Login()
{
    return "Форма для входа";
}

[HttpPost]
public string Login(LoginModel model)
{
    return "Форма для входа";
}
```

Здесь показаны два метода C# с одним и тем же именем, но они привязаны к разным HTTP-методам. Это возможно, потому что у них разные параметры, хотя имена методов и одинаковые.

Но даже если методы будут получать одинаковые параметры, мы все же можем указать одновременно два атрибута:

```
[HttpGet]
[Route("Login")]
public string Login()
{
    return "Форма для входа";
}
```

```
[HttpPost]
[Route("Login")]
public string LoginSave()
{
    return "Форма для входа";
}
```

На этот раз имена методов разные, но за счет того, что они через атрибуты привязаны к одному URL, но к разным HTTP-методам, это решит нашу задачу.

В разд. 13.11 мы познакомимся с этим на практике.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter13\AttributeRouting* сопровождающего книгу электронного архива (см. *приложение*).

13.6. Подробнее про контроллеры

Итак, мы научились с помощью различных URL вызывать код в приложении, и теперь пора перейти к вопросу отображения данных.

До сих пор в качестве результата я всегда возвращал просто строку. Методы контроллера объявлялись так, чтобы они отображали строку, и в качестве результата возвращалась она же. Это работает потому, что .NET пытается отобразить любые возвращаемые данные.

Но это такая вспомогательная функция, которой лучше не злоупотреблять, и, честно говоря, я ее использую только в обучающих целях. Желательно все же, чтобы методы контроллера возвращали в результате любой класс, который реализует интерфейс `IActionResult`:

```
public IActionResult Index()
{
    return Content("Test");
}
```

Здесь я объявил метод, который возвращает более подходящий для сетевых операций результат: `IActionResult`. Опять же, пример будет простой, и чтобы вернуть данные, которые соответствуют этому интерфейсу, я пользуюсь простым методом `Content`. Метод `Content` нам достается за счет того, что мы наследуем класс `Controller`.

Метод `Content` получает на входе строку и возвращает в качестве результата объект класса `ContentResult`, который, в свою очередь, является реализацией `IActionResult`.

Таким образом, мы сделали практически то же самое, что и в предыдущем разделе, — отображали на странице просто текстовую строку, но сделали это в соответствии с рекомендациями.

Если посмотреть на «внутренности» решения, то мы увидим, что `ContentResult` происходит от класса `ActionResult`:


```
public class ContentResult : ActionResult,
    IActionResult,
    IStatusCodeActionResult
```

Класс `ActionResult` является абстрактным, и его цель — реализовать нужный интерфейс `IActionResult`:

```
public abstract class ActionResult : IActionResult
```

На основе базового класса `ActionResult` в `.NET Core/.NET 7` создано несколько классов, которые предназначены для разных типов возвращаемых данных, и все они находятся в пространстве имен `Microsoft.AspNetCore.Mvc`:

☐ <code>ContentResult</code>	☐ <code>RedirectResult</code>
☐ <code>EmptyResult</code>	☐ <code>SignInResult</code>
☐ <code>FileResult</code>	☐ <code>SignOutResult</code>
☐ <code>ForbiddenResult</code>	☐ <code>StatusCodeResult</code>
☐ <code>JsonResult</code>	☐ <code>ViewComponentResult</code>
☐ <code>LocalRedirectResult</code>	☐ <code>ViewResult</code>
☐ <code>ObjectResult</code>	☐ <code>RazorPages.PageResult</code>
☐ <code>PartialViewResult</code>	

То есть теоретически в своем коде вы можете вместо `IActionResult` возвращать базовый класс `ActionResult`, и от этого функциональность не поменяется, но все же лучше использовать интерфейс. Разница между привязкой к интерфейсу и к абстрактному классу небольшая, но привязка к интерфейсам с точки зрения правильного кода лучше, чем привязка к классам, даже абстрактным.

Следующий код тоже вполне легален, но, повторю, желательно все же использовать интерфейс:

```
public ActionResult Index()
{
    return Content("Test");
}
```

Так как наш контроллер происходит от класса `Controller`, вместе с ним мы получаем много интересной функциональности, которая может пригодиться. Например, если обратиться к дескриптору:

```
this.ControllerContext.ActionDescriptor
```

то в нем мы найдем много полезной информации — такой, в частности, как имя текущего выполняемого действия `Action`.

Но чаще всего вам будет необходимо получить доступ к HTTP-контексту. Раньше это было глобальное свойство, которое оказывалось сложно перезаписать, и из-за него возникали проблемы с юнит-тестами и интеграционными тестами. Сейчас это свойство контроллера, и к нему можно получить доступ через `this.HttpContext`. При этом в свойстве `this.HttpContext.Request` будет содержаться вся информация

о запросе: о том, какой URL пользователь вызвал, какие параметры передал, какой тип запроса пришел: GET или POST, и т. д.

У контроллера есть еще несколько интересных методов, способных генерировать специальные типы. Например, у нас может быть метод, который показывает на странице какую-либо статью, а номер статьи должен передаваться в качестве параметра. Если по номеру никакой статьи не найдено, то мы можем вернуть стандартную ошибку 404, вызвав метод контроллера `this.NotFound`:

```
public IActionResult loadarticle(int id)
{
    if (id == 10) {
        return Content("Good stuff");
    }
    return this.NotFound();
}
```

А если обратиться к URL этого метода и указать неверный ID, то мы также увидим классическую ошибку 404.

Это классический метод обработки классической проблемы. И если вам кажется, что страница ответа выглядит очень уж страшной, то это поправимо — есть способ сделать ее красивее, но это тема отдельного разговора.

Для тех случаев, когда переданы некорректные данные, нужно использовать метод `BadRequest`. Например, номер статьи не должен быть меньше нуля, а значит, мы можем добавить проверку:

```
if (id < 0)
{
    return this.BadRequest("Something is really bad");
}
```

Благодаря тому, что мы возвращаем интерфейс `IActionResult`, мы можем возвращать как реальный контент, так и ошибки `BadRequest` или `NotFound`. Если бы мы имели в виду возвращать строку, то при отсутствии ошибок строку бы и возвращали, но в случае ошибки приходилось бы использовать обходные пути. А `IActionResult` позволяет получить элегантное решение на все случаи жизни.

Есть еще метод `File`, с помощью которого можно передать пользователю с сервера файл.

А можем мы вернуть в качестве результата метода что-то совсем «сумасшедшее»? Да! Давайте создадим класс `Person` и попробуем вернуть объект этого типа:

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Теперь создаем новый метод в контроллере:

```
public Person person()
{
    return new Person {
        FirstName = "Михаил",
        LastName = "Фленов"
    };
}
```

Если обратиться к этому методу, то браузер отобразит объект в виде JSON:

```
{"firstName":"Михаил","lastName":"Фленов"}
```

И хотя возвращать класс произвольного типа можно, я не устану повторять, что лучше использовать `ActionResult`, а чтобы объект превратить в такой интерфейс, можно использовать класс `ObjectResult`:

```
public ActionResult person()
{
    return new ObjectResult(new Person {
        FirstName = "Михаил",
        LastName = "Фленов"
    });
}
```

Результат будет тот же самый.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\ActionResult` сопровождающего книгу электронного архива (см. приложение).

13.7. Представления

Возвращать пользователю код страницы в виде текста можно, но сложно. Самый популярный и эффективный метод отобразить страницу — использовать представления. До сих пор мы отображали что-либо простое с помощью контроллеров, но отображать так целую страницу может быть далеко не самым эффективным решением.

Контроллеры, которые мы рассматривали в предыдущем разделе, должны принимать только базовые решения, а вся логика должна жить отдельно. Контроллеры также не должны отвечать и за отображение данных — это дело *представлений*, и это я уже отмечал в *разд. 13.4*.

Для создания представлений сейчас самым популярным движком является `Razor`. Это встроенный движок, но можно создать и свой собственный, хотя я никогда не видел в этом надобности.

Когда контроллер хочет отобразить специальный файл представления, он должен вернуть в качестве результата работы метода тип `ViewResult`, который можно создать, просто вернув результат работы метода `View`:

```
public IActionResult Index()
{
    return View();
}
```

У нас в проекте уже есть два контроллера — давайте добавим для тестирования еще один: `RazortestController`:

```
using Microsoft.AspNetCore.Mvc;

namespace MyWebSite
{
    public class RazortestController : Controller
    {
        public IActionResult Index()
        {
            return View();
        }
    }
}
```

Так как контроллер называется `Razortest`, а метод — `Index`, то, согласно маршрутизации по умолчанию, доступ к такому методу можно получить из браузера. Но если обратиться по адресу: `/razortest/index`, произойдет ошибка (рис. 13.6), потому что нет самого представления, которое должно быть отдельным файлом, где мы можем писать теги HTML и инструкции C# с помощью синтаксиса Razor.

Тут нам говорят, что представление `View` с именем `Index` не найдено. А пытались его найти в следующих местах: `/Views/Razortest/Index.cshtml` и `/Views/Shared/Index.cshtml`.

При этом поиск происходит в указанном порядке. Первое, куда смотрит система, — в папку `View`. Потом — в папку с таким же именем, как у контроллера (кроме самого слова `Controller`). У нас контроллер называется `RazortestController`, так что, отбросив слово `Controller`, путь превращается в `/Views/Razortest`. И вот здесь должен присутствовать файл с именем, которое совпадает с именем метода и с расширением `cshtml`. У нас метод `Index`, а значит, файл должен нести имя `Index.cshtml`.

Расширение имени файла `cshtml` не случайно — это сочетание `cs` и `html` и отражает реальную суть файла: здесь можно смешивать код HTML и код CS.

Итак, теперь, когда мы обращаемся к контроллеру, он вызывает метод `View`, который находит файл представления, исполняет в нем код и возвращает результат пользователю в виде HTML.

Имя CSHTML-файла можно изменить. Если вы хотите в методе `Index` отобразить, например, файл `Test.cshtml`, это можно сделать, если указать имя нужного нам представления в качестве параметра метода `View`:

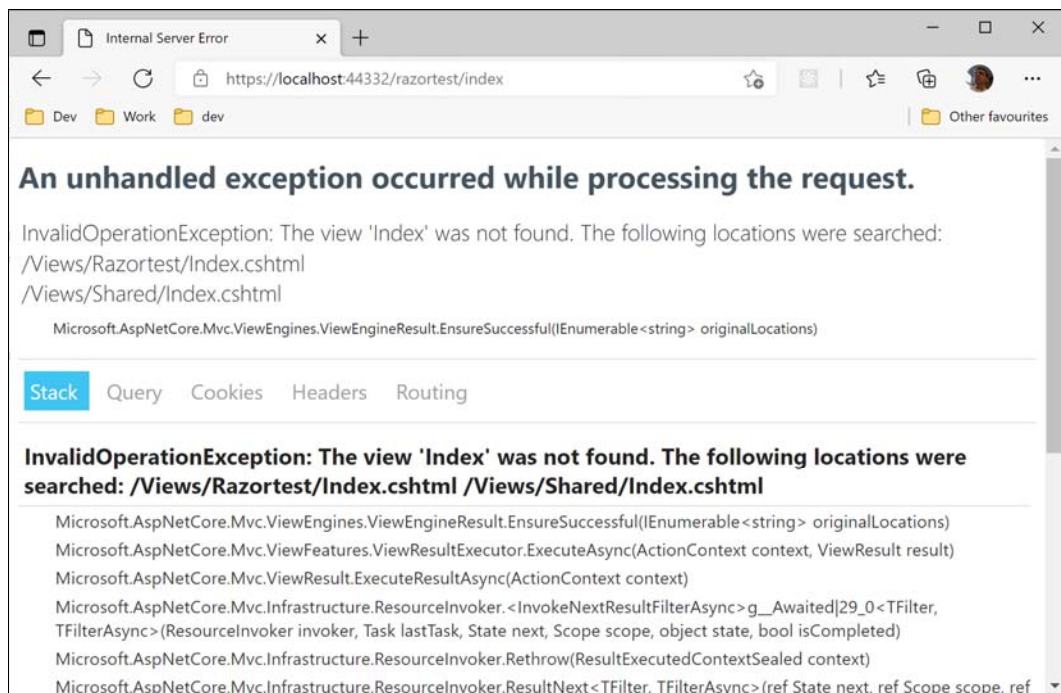


Рис. 13.6. Ошибка: не найден файл представления

```
public IActionResult Index()
{
    return View("Test");
}
```

Однако этот метод также завершит выполнение ошибкой, потому что представление с именем `Test` у нас тоже нет.

В тех случаях, когда представление будет использоваться только для определенного контроллера, его желательно помещать в папку с именем контроллера — в нашем случае это `/Views/Razortest`. Если же представление предполагается использовать сразу с несколькими разными контроллерами или действиями, то его рекомендуется помещать в папку `Shared`.

Для первого примера мы создадим представление, которое будет обслуживать только один метод, и поэтому вернемся к имени по умолчанию:

```
public IActionResult Index()
{
    return View();
}
```

Если у вас в проекте еще нет папки `Views`, то создайте ее, а внутри нее — вложенную папку `Rasortest`. Обе папки можно создать в файловой системе, используя файловый менеджер, или с помощью среды разработки `Visual Studio`: щелкните правой

кнопкой мыши на папке **Views**, выберите из контекстного меню **Add | New Folder** и введите имя папки: **Rasortest**.

Теперь нужно создать собственно представление **View** — щелкните правой кнопкой мыши на папке **Rasortest**, выберите **Add | New File**, и перед нами откроется уже знакомое окно выбора типа и имени файла. Выберите **Razor View — Empty**, как показано на рис. 13.7, а в качестве имени файла оставьте **Index.cshtml**.

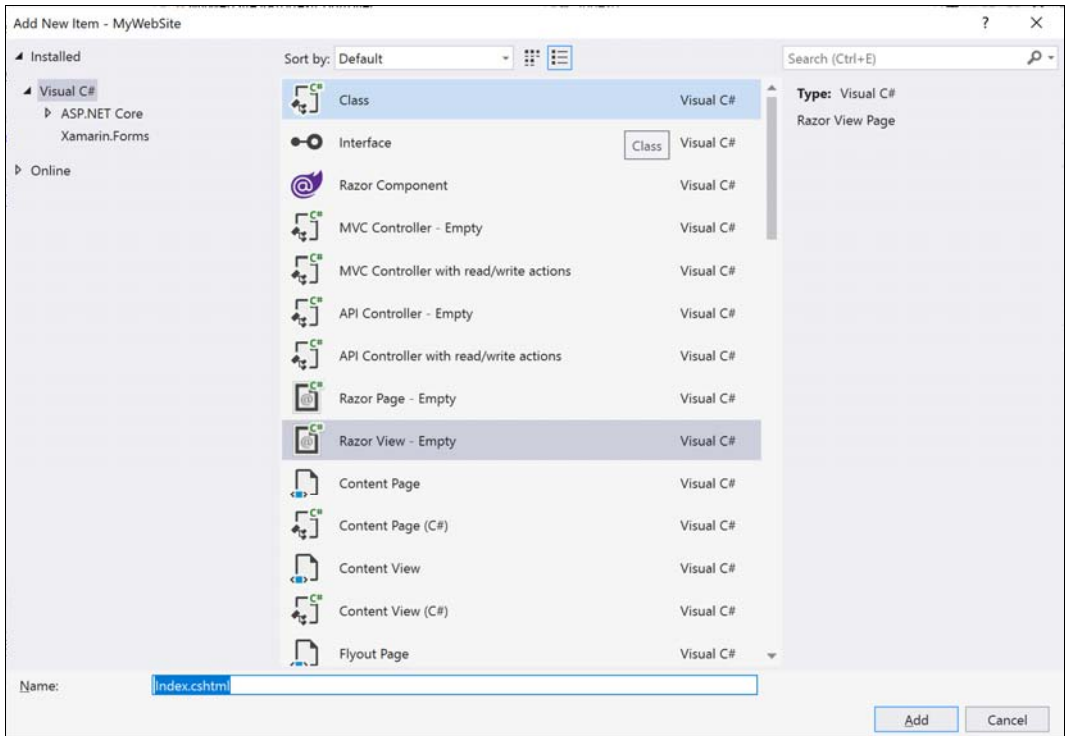


Рис. 13.7. Создаем файл представления

Теперь все представления, которые мы станем использовать в контроллере **RazorController**, будут храниться в папке **Views/Razortest**. Убираем все, что мастер поместил в созданный файл **Index.cshtml**, и вместо этого напишем немного HTML-кода:

```
<html>
  <body>
    <h1>Hello</h1>
    <p>This is a view</p>
  </body>
</html>
```

Запустите сайт, перейдите по адресу **/razortest/index**, и если вы правильно создали папки для хранения файла **Index.cshtml** и не ошиблись в имени файла, то должны увидеть страницу, показанную на рис. 13.8.

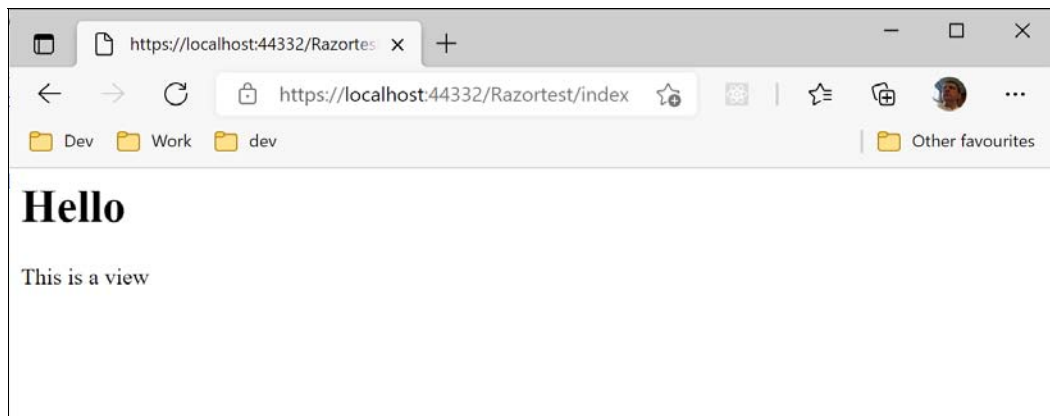


Рис. 13.8. Страница представления

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter13\Razortest* сопровождающего книгу электронного архива (см. приложение).

13.8. Модель представления

В веб-программировании выделяют бизнес-модели и модели представления. Мы уже слегка касались вопроса бизнес-моделей, в которых нужно реализовывать логику сайта, но эту тему мы пока немного отложим. А вот модели представления уже пора начать использовать.

В MVC представления не должны ничего знать о бизнес-модели, и бизнес-модель ничего не должна знать о том, как и кем будут отображаться данные. Вся коммуникация должна происходить через контроллер, который единственный знает о том и о другом.

Но как представление может получить доступ к данным? Оно же не может вызвать метод бизнес-логики и увидеть ответ? Вместо этого представления должны получать от контроллера все необходимые данные в виде *моделей представления*. Модели представления — не какой-то код с логикой, это могут быть классы, которые описывают лишь модель данных и могут еще заниматься проверкой данных на корректность, и всё. В рассматриваемом случае слово «модель» предполагает именно моделирование. Если под бизнес-моделью часто понимают не только данные, но и логику, то тут — только данные. Попросту говоря, это классы, которые содержат данные для отображения в представлении или для их передачи логике приложения.

Для хранения моделей представления я предпочитаю использовать отдельную папку с именем **ViewModel** — добавьте ее через файловую систему или Проводник решения.

Затем в папке модели создайте файл **Person.cs**. Для этого щелкните правой кнопкой мыши на папке **ViewModel** и выберите **Add | New File**. В открывшемся окне выбо-

рите раздел **General** и потом **Empty Class**. Внизу окна введите `Person.cs` и нажмите кнопку **ОК**.

Напоминаю, что какой тип файла вы выберете — не имеет значения, главное — это расширение файла и его содержимое. А вот содержимым мы сейчас и займемся.

Этот файл создан специально для модели, и, судя по его названию, здесь я хочу создать класс с именем `Person` и несколькими полями. Пусть для примера у нашего «персонажа» будут три свойства: имя, фамилия и возраст:

```
namespace MyWebSite.ViewModel
{
    public class Person
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public int Age { get; set; }
    }
}
```

Я уже использовал в этой главе такой класс, но тогда я его создавал прямо в файле с контроллером, что очень плохо. Теперь я поступил правильно и поместил класс в отдельную папку для моделей представления.

У нашей модели есть лишь свойства, потому что ее смысл — только создать объект в памяти и через контроллер передать данные в представление.

Итак, идем в контроллер `RazertestController`, здесь в методе `Index` создаем объект класса `Person` и, чтобы передать его представлению, просто передаем его в качестве параметра методу `View`:

```
public IActionResult Index()
{
    Person person = new Person()
    {
        FirstName = "Mikhail",
        LastName = "Flenov",
        Age = 42
    };

    return View(person);
}
```

Так как мы поместили модели представления в отдельную папку, то по умолчанию для них среда разработки, скорее всего, задала свое пространство имен. В нашем случае это `MyWebSite.ViewModel`, поэтому не забудьте подключить его в контроллере.

Пока мы использовали `View` без параметров — тогда фреймворк просто искал файл с таким же именем, как и имя метода, и отображал его. Я упоминал, что есть еще версия, когда в качестве параметра передается имя представления, если нужно отобразить файл, имя которого отличается от имени метода.

В этом разделе мы познакомимся с еще двумя версиями этого метода:

- ❑ `View(object model)` — в качестве параметра передается объект модели представления. При этом фреймворк будет искать файл представления с таким же именем, как и имя метода;
- ❑ `View(string viewName, object model)`. Первый параметр — это имя файла представления, а второй — объект модели. Через модель мы будем передавать данные, которые нужно отобразить на странице в представлении.

Запомнить это просто: если используется представление с таким же именем, как и имя метода, то модель — это первый и единственный параметр. Если нужно указать имя представления явно, то модель будет во втором параметре.

Теперь осталось увидеть, как в представлении можно использовать модель. Итак, смотрим на обновленный код представления `Views/Razertest/Index.cshtml`:

```
@model MyWebSite.ViewModel.Person

<h1>Hello</h1>
<p>First Name: @Model.FirstName</p>
<p>Last Name: @Model.LastName</p>
<p>Age: @Model.Age</p>
```

В первой строке мы указываем класс, который ожидается увидеть в представлении. Это необходимо по двум причинам: чтобы защититься от передачи неверной модели и чтобы среда разработки могла подсказывать через автозаполнение свойства модели:

```
@model MyWebSite.ViewModel.Person
```

Обратите внимание, что здесь указывается полное имя класса вместе с пространством имен, потому что наше представление пока не знает, в каких пространствах искать.

В определении метода `View` параметр модели имеет тип `Object`, а через этот тип в .NET можно передать совершенно любой тип данных, — ведь все происходит именно от класса `Object`. Но в самом файле представления нам нужно более точно указать, какого именно класса будут данные, чтобы фреймворк знал, какие именно свойства есть у модели, и мы могли проще к ним обращаться.

Теперь, когда мы хотим вывести на экран страницу со значением какого-либо свойства, достаточно указать `@Model` и затем свойство модели. Например, имя мы выводим так: `@Model.FirstName`. При отображении представления фреймворк подставит вместо `@Model` объект модели, который передается в качестве параметра, и на странице будет показано имя «персонажа».

Запустите пример и в браузере перейдите по адресу: <https://localhost:7261/razortest/index>. Результат работы этого примера показан на рис. 13.9.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\ViewModel` сопровождающего книгу электронного архива (см. приложение).

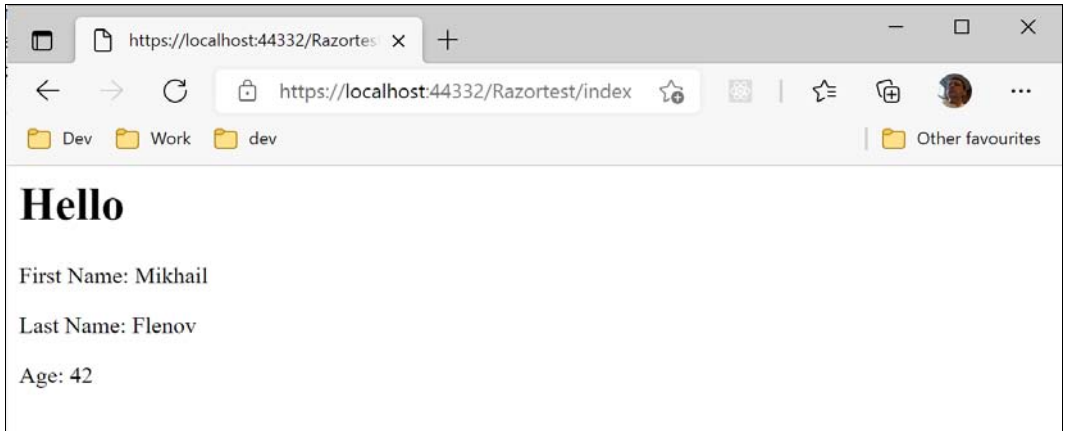


Рис. 13.9. Отображение данных представления

13.9. Представления Razor

Синтаксис программирования Razor — это современный подход к созданию представлений в приложениях ASP.NET. Если бы нам нужны были статичные HTML-файлы, мы бы не заморачивались с .NET Core/.NET 7 или вообще с каким-либо языком программирования, а просто создали бы статичные файлы и отображали их. Но такие сайты уже мало кому нужны — сейчас даже простые страницы требуют динамики, и содержимое их постоянно меняется.

С помощью Razor мы можем писать логику в представлениях. Это всего лишь оформление логики, так что совершенно новый язык нам изучать не придется — достаточно будет только познакомиться с тем, как можно использовать в представлениях уже знакомый нам C#.

13.9.1. Комментарии

Razor — это как новый язык, а ознакомление с новыми языками начинают со знакомства с комментариями. Как и в любом другом языке программирования, комментарии не выполняются фреймворком, не влияют на результат и даже не отображаются на результирующей странице.

Комментарии в представлениях Razor пишутся между символами `@*` и `*@`, например, так:

```
@*  
    Это многострочный комментарий  
    в Razor и веб-приложении .NET Core/.NET  
*@
```

Я противник комментариев, и в моем коде вы встретите их очень редко, потому что код должен быть таким, чтобы его можно было читать без каких-либо дополнительных подсказок. Комментарии необходимы только в самом крайнем случае.

Очень часто программисты используют комментарии для того, чтобы временно отключить какую-либо функциональность. Я тоже иногда так поступаю, но только уж совсем для временного отключения. То же самое можно делать и в представлении. Причем комментарием становится совершенно всё, включая код HTML:

```
@*
<h1>@Model.LastName</h1>
*@
```

Здесь внутри комментария есть и HTML-код, и даже немного Razor-кода. Ни то ни другое на странице не отобразится. Но если HTML-комментарии небезопасны, потому что их можно увидеть в исходнике страницы, то Razor-комментарии безопасны — их в исходнике видно не будет.

Повторюсь для убедительности: даже несмотря на безопасность, использовать комментарии рекомендуется только для временного отключения какой-либо функциональности и лишь в крайних случаях.

13.9.2. Вывод данных в представлениях

Представления в .NET — это файлы *.cshtml, и их расширение не просто так состоит из двух частей:

- cs, или C Sharp, или просто C#;
- html, а он и в Африке HTML.

Первая часть — это указание на язык программирования, а вторая — указание на язык разметки. В результате мы получаем смесь, в которой можем писать и то и другое. Описание HTML выходит за рамки нашего разговора, и я предполагаю, что вы с ним уже знакомы.

Мы уже знаем, что представления могут получать на входе какую-то модель, и мы даже пробовали выводить что-то в *разд. 13.8*, посвященном модели представления. Чтобы вывести какие-то данные из модели на страницу, перед переменной C# нужно просто поставить символ @. Напоминаю: для вывода поля `FirstName` в нашей модели мы пишем так: `@Model.FirstName`.

Это всё мы уже видели, а теперь двинемся дальше. Бывает, что необходимо вывести больше, чем просто какую-то простую переменную, и в этом случае после символа @ мы заключаем последующий текст в круглые скобки: `@(значение)`.

Например, если в свойстве модели находится число, то мы можем выполнить в этом выражении какую-нибудь математическую операцию:

```
@(Model.Age * 2)
```

Здесь я вывожу возраст «персонажа» и зачем-то умножаю его на 2. Просто не придумал примера получше...

Со строковыми переменными тоже можно выполнять какие-либо операции — например, прибавить какой-то текст, а имя отобразить в нижнем регистре:

```
@("Mr. " + Model.FirstName.ToUpper())
```

Прибавлять "Mr." в круглые скобки смысла нет — его можно было просто вывести до @() так:

```
Mr. @Model.FirstName.ToUpper()
```

но мне захотелось показать какую-нибудь операцию, которую можно выполнить внутри @(), и лучше варианта я снова не нашел.

Тут можно написать даже просто какой-либо C#-код — главное, чтобы он возвращал что-либо, что и будет отображаться на странице. Например, здесь мы с помощью `Int32.Parse` приведем строку к числу, и оно будет отображено на странице:

```
Число: @(Int32.Parse("4"))
```

Итак, просто для вывода переменной на страницу достаточно поставить перед этой переменной символ @. Если в выводе имеются пробелы или он представляет собой нечто большее, чем одна переменная (как в случае с умножением возраста), надо заключить всё в скобки.

13.9.3. C#-код

Если мы посмотрим в представление, которое для нас сгенерировал Visual Studio, то, возможно, у вас в нем будут присутствовать две «магические» строки, которые стоит рассмотреть:

```
@{  
}
```

«Магический» символ @ переводит фреймворк в режим выполнения кода, а фигурные скобки указывают его границы:

```
@{  
    int i = 10;  
}
```

В этом примере в режиме кода создается переменная `i`, которой присваивается значение 10. Ничего особенного, просто что-то нужно было написать для примера.

Совет

В режиме кода можно писать практически все что угодно — никаких ограничений вроде бы нет. Можно даже полностью отказаться от контроллеров и обращаться к модели прямо из View, потому что никто вас остановить не может, если нет наставника или начальника, который даст по рукам. Однако в представлениях не должно быть никакой логики! Да, ее туда включить возможно, но это очень плохо. В файлах `*.cshtml` может быть только такой C#-код, если он необходим для отображения чего-либо на странице.

У начинающего программиста может возникнуть вопрос: что писать в контроллерах, а что — в `*.cshtml`? Трудно дать такое правило, которое работало бы в 100% случаев. Я бы сказал так: если код можно написать где-то в другом месте, в представлении лучше его не писать. В представлении должна быть только логика отображения.

Как только фигурные скобки заканчиваются, мы снова возвращаемся в режим HTML и можем писать текст даже без тегов:

```
@{
    int i = 10;
    <p>index = @i</p>
}
<p>Это просто текст, а не код</p>
```

В этом примере я еще и HTML-код вставил в блок кода, и это совершенно легально. Как только фреймворк видит открывающийся HTML-тег, он переходит в режим отображения этого языка разметки. Как только достигается конец тега, то мы возвращаемся в режим кода и можем писать C#-код. Написать просто текст среди кода нельзя, но это и не нужно. В тех случаях, когда вам нужно в блоке кода вывести на страницу какой-то текст, можно использовать тег `<text>`:

```
@{
    int i = 10;
    <text>index = </text>@i
}
```

Это очень удобный трюк, и когда фреймворк отображает содержимое тега `<text>`, то на странице не будут отображаться никакие пробелы до открывающего тега и после него, так что можете ставить пробелы так, чтобы код выглядел в редакторе красиво.

Есть еще один вариант перевести текущую строку в HTML-режим — поставить `@:` (символ `@` и двоеточие):

```
@{
    int i = 10;
    @: This is a string @i
}
```

В этом примере у нас в блоке кода первая строка — это чистый C#-код, а вторая — переведена в режим HTML, поэтому здесь можно писать просто текст, а для вывода значения переменной `i` мне приходится отображать ее, как это делается в HTML-режиме, т. е. ставить перед ней символ `@`.

13.9.4. Условные операторы

При отображении представления очень часто необходимо указывать какую-то логику, отображать данные в зависимости от каких-либо условий — в общем, использовать оператор `if` или `switch`.

Как и с выводом данных, все начинается с «магического» символа `@`, после которого идет оператор `if` без каких-либо отступов. То есть символ `@` как бы переводит систему в режим C#-кода, и парсер начинает искать C#-инструкции:

```
@if (Model.Age < 15)
{
    <p>Классно быть молодым, но сайт только для лиц старше 15 лет</p>
}
```

```
else
{
    <p>Все отлично</p>
}
```

Понимаю, что пример этот, может быть, не самый логичный, но, главное, наглядный. Как видите, символ `@` необходим только перед оператором `if`. Перед `else` ничего ставить не нужно, потому что он должен идти сразу же после закрывающей фигурной скобки. Если `else` нет, а идут какие-нибудь HTML-теги или просто текст, то фреймворк возвращается к отображению HTML и не воспринимает больше последующий текст как C#-код.

То есть конструкция `@if {}`, как и конструкция `@{}`, переводит парсер в режим кода, и все, что находится внутри фигурных скобок, по умолчанию будет восприниматься как C#-код. Впрочем, внутри фигурных скобок мы можем также перейти как бы в режим HTML (я не знаю, правильно ли будет говорить «режим»?), когда все воспринимается как текст, — если указать теги. В нашем случае парсер файлов `*.cshtml` видит, что это точно текст, потому что он обрамлен в HTML-теги, и начинает этот текст выводить.

Как только закончится текущий HTML-тег, фреймворк возвращается в режим выполнения кода и ищет C#-инструкции или закрывающую фигурную скобку. Закрывающая фигурная скобка тоже является частью C#-кода, поэтому переход в режим кода вполне логичен:

```
else
{
    <p>отлично</p>
    <pre>int index = 10;</pre>
    <p>index = @index</p>
}
```

Здесь в `else` присутствуют три строки HTML-кода. По окончании HTML-кода первой строки начинается C#-код, который корректен, и его можно выполнить. Третья строка — снова HTML-код, потому что начинается с корректного тега.

Вот так HTML-теги переключают фреймворк с режима выполнения кода на режим отображения HTML и обратно. В тех случаях, когда среди HTML-кода нам нужно выполнить C#-код, мы всегда можем сделать это, написав:

```
@{
    // C#-код
}
```

В большинстве языков программирования имеется конструкция `else if` — чтобы было больше двух вариантов выбора, и в Razor это тоже возможно:

```
@if (Model.Age < 15)
{
    <p>Классно быть молодым, но сайт только для лиц старше 15 лет</p>
}
```

```
else if (Model.Age < 18)
{
    <p>Все возможно, но с ограничениями</p>
}
else
{
    <p>Все отлично</p>
}
```

Здесь у нас выводятся разные сообщения — в зависимости от того, какой возраст у человека: до 15 лет, от 15 до 18 или старше 18.

В Razor также есть возможность использовать оператор `switch`, который работает идентично его C#-аналогу:

```
@switch (value)
{
    case 1:
        <p>Это первый случай</p>
        break;
    case 2:
        <p>Это второй случай</p>
        break;
    default:
        <p>Ну и случай по умолчанию</p>
        break;
}
```

Не думаю, что здесь нужны какие-либо дополнительные комментарии для тех, кто уже знаком с C#.

13.9.5. Циклы

Трудно сказать, что популярнее при работе с представлениями — логические операции или циклы, но они точно играют очень важную роль. Хотя я оба эти приема поставил бы на вторую строку, а первую отдал бы выводу информации.

В Razor поддерживаются все необходимые нам возможности работы с циклами: `for`, `foreach`, `while` и `do..while`. Они работают так же, как и в коде на C#, просто в представлении обращение к ним нужно начинать с символа `@`, как мы уже делали в предыдущем разделе с логическими операциями. При этом для циклов я буду использовать контроллер `RazortestController`. Итак, несколько примеров.

Создадим новый метод `list`:

```
public IActionResult List()
{
    var people = new Person[] {
        new Person() { FirstName="Mikhail", LastName="Flenov", Age=42 },
        new Person() { FirstName="Ivan", LastName="Sergeev", Age=29 },
    }
```

```

        new Person(){ FirstName="Lena", LastName="Petrova", Age=20 },
    };
    return View(people);
}

```

Здесь в методе `List` из трех объектов класса `Person` создается массив, который помещается в переменную `people`. После этого мы возвращаем в качестве результата метод `View` и передаем ему в качестве модели наш массив.

Поскольку я не указал, какое представление отображать, то фреймворк будет искать по умолчанию файл `List.cshtml` в папке `Views\Razortest`. Создайте такой файл и давайте начнем писать в нем код. В самом начале нужно указать модель, которую мы получили:

```
@model MyWebSite.ViewModel.Person[]
```

Теперь на примере этой модели можно рассмотреть все возможные циклы. Первый — классический `for`:

```

<h1>for</h1>
<p>Цикл for:</p>
<ul>
    @for (var i = 0; i < Model.Length; i++)
    {
        <li>@Model[i].LastName @Model[i].LastName (@Model[i].Age)</li>
    }
</ul>

```

Затем более популярный `foreach`, который, на мой взгляд, удобнее для перебора элементов массива:

```

<h1>foreach</h1>
<p>Цикл foreach:</p>
<ul>
    @foreach (var p in Model)
    {
        <li>@p.LastName @p.LastName (@p.Age)</li>
    }
</ul>

```

Следующий цикл — `while`. Я не очень часто его использую, потому что он достаточно специфичный:

```

<h1>while</h1>
<p>Цикл while:</p>
<ul>
    @{
        int index = 0;
    }
    @while (index < Model.Length)
    {
        <li>@Model[index].LastName
            @Model[index].LastName (@Model[index].Age)</li>
    }

```



```
        index++;  
    }  
</ul>
```

В приведенном примере для цикла `while` нужна какая-нибудь переменная, которую мы будем использовать как счетчик, и именно из-за этого я больше предпочитаю другие циклы. Здесь создается переменная `index` типа `int` прямо перед блоком цикла. Так как это должна быть C#-переменная и это должен быть C#-код, его я написал в блоке `@{ }`:

```
@{  
    int index = 0;  
}
```

Ну и, наконец, цикл `do..while`. Его используют, наверное, реже всего:

```
<h1>do..while</h1>  
<p>Цикл do..while:</p>  
<ul>  
    @{  
        index = 0;  
    }  
  
    @do  
    {  
        <li>@Model[index].LastName  
            @Model[index].LastName (@Model[index].Age)</li>  
  
        index++;  
    } while (index < Model.Length);  
</ul>
```

Так как в предыдущем цикле я уже создал переменную `index`, а в этом хочу просто использовать ее заново, перед началом цикла нужно установить значение переменной в 0, поэтому и написан такой блок:

```
@{  
    index = 0;  
}
```

Здесь уже созданной ранее переменной `index` присваивается значение 0, чтобы начать цикл с начала.

Не думаю, что нужно останавливаться на каком-либо из циклов отдельно, потому что мы сейчас не учимся программировать, и я полагаю, что базовые знания C# у вас имеются. Здесь все работает абсолютно так же, как в большинстве C-подобных языков. Кроме того, все эти циклы подробно рассмотрены в *разд. 2.7* и *2.8*.

А если вспомнить, что символ `@` и фигурные скобки переводят представление в режим кода, то вы можете писать в цикле любой C#-код:

```
@{  
    for (int i = 0; i < 10; i++) {  
    }  
}
```

Здесь перед оператором `for` нет символа `@`, потому что мы уже находимся в режиме кода и используем циклы в классическом `C#`-стиле.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\loops` сопровождающего книгу электронного архива (см. приложение).

13.9.6. Исключительные ситуации

Это случается не так часто, но бывает необходимость отлавливать исключительные ситуации даже в представлениях. Как я уже говорил, в представления не следует включать никакую бизнес-логику — они должны только отображать данные, и поэтому вероятность возникновения проблем в них должна быть очень низкой.

Я не смог придумать ни единого хорошего примера для демонстрации исключительной ситуации в представлении, потому что любые расчеты должны происходить в модели или, в крайнем случае, в контроллерах.

Если же вы столкнулись с ситуацией, когда какой-либо `C#`-код в представлении может вызывать исключительную ситуацию, то ее можно поймать так:

```
@try {
    // код, который может сгенерировать ошибку
}
catch (Exception e) {
    <p>Произошла ошибка</p>
}
finally {
    <p>Мы попали в finally</p>
}
```

13.9.7. Ключевое слово *using*

Когда модель представления, которую мы получаем из внешнего источника, содержит какие-либо объекты, для обращения к их классам придется писать в представлении их полное название. Например:

```
@foreach (MyWebSite.ViewModel.Person p in Model)
{
    <li>@p.LastName @p.LastName (@p.Age) </li>
}
```

Я тут решил явно указать тип переменной `MyWebSite.ViewModel.Person` в цикле, хотя большинство сейчас просто использует `var`. Оба варианта отлично работают, но если вы не хотите каждый раз писать `MyWebSite.ViewModel.Person`, то в представлении можно подключить пространство имен так же, как и в `C#`-коде, — с помощью ключевого слова `using`, просто в представлении перед ним нужно поставить символ `@`:

```
@using MyWebSite.ViewModel
@foreach (Person p in Model)
```

```
{  
    <li>@p.LastName @p.LastName (@p.Age) </li>  
}
```

В C# ключевое слово `using` может использоваться в двух случаях: для подключения пространств имен и для указания области видимости для объектов классов, которые реализуют интерфейс `IDisposable`. Здесь та же песня, только с применением символа `@`:

```
@using (Html.BeginForm()) {  
    // Здесь будет форма  
}
```

Это классический способ использования блока `@using` в представлениях с интерфейсом `IDisposable`. Обратите внимание: я здесь применил HTML-помощник `Html.BeginForm()`. Надо отметить, что HTML-помощники содержат методы, которые прячут HTML-код от программистов. Честно говоря, мне такой подход не очень нравится, но это мое личное мнение: я предпочитаю чистый HTML и считаю, что в представлении должна быть минимальная привязка к C#-коду.

Однако рассмотрим этот код, и что он делает. Метод `BeginForm` HTML-помощника `Html` выведет на страницу тег `<form>` и вернет в качестве результата объект класса `MvcForm`, который в недрах фреймворка объявлен следующим образом:

```
public class MvcForm : IDisposable
```

У интерфейса `IDisposable` есть метод `Dispose()`. Работает он так: когда объект типа `MvcForm` уже не нужен системе, то вызывается метод `Dispose()`, который проверяет, закрыт ли открытый тег `<form>`, и если нет, то закрывает его, для чего выводит на страницу тег `</form>`. Другими словами, когда мы выйдем за границы видимости `@using`, система закроет форму, и в HTML-коде мы увидим:

```
<form>  
    // Здесь будет форма  
</form>
```

На самом деле там будут еще атрибуты `action` и `method`, а также токен безопасности. Пока опустим это, потому что разговор идет именно о `@using`.

Без ключевого слова `using` то же самое можно было бы сделать так:

```
@{var form = Html.BeginForm();}  
// Здесь будет форма  
@{form.EndForm();}
```

Здесь я явно открыл и закрыл форму методами `BeginForm` и `EndForm` соответственно. Как мы уже знаем, метод `BeginForm` возвращает объект, который я сохранил в переменной `form`, и когда форма уже не нужна, я вызываю метод `EndForm` этого объекта.

Какой из этих методов вы выберете — это уже личные предпочтения, но, на мой взгляд, использование ключевого слова `using` делает код чище и красивее. Впрочем, и в других случаях наличия интерфейса `IDisposable` лучше все же задейство-

вать именно `using` не только с эстетической точки зрения, но и с точки зрения эффективности результата.

13.10. Создание представлений

Мы уже умеем указывать, какое именно представление будет отображаться из контроллера как реакция на различные действия, но это только малая часть того, что мы можем делать. Настала пора более подробно поговорить о представлениях.

Если посмотреть на большинство сайтов, то у них от страницы к странице повторяются одни и те же элементы. У большинства это как минимум шапка и подвал. Помимо них, там еще могут присутствовать меню, рекламные блоки, панели слева и/или справа.

Если в коде копировать повторяющиеся части от страницы к странице, то это будет нереальный ужас. Проблема решается двумя способами: использованием макетов (шаблонов) или выносом повторяющихся участков в отдельные файлы и подключением их по мере надобности. В `.NET Core/.NET 7` поддерживаются и макеты, и подключение файлов.

Основа сайта строится через шаблон, который могут наследовать другие страницы. Внутри страниц шаблон наследуется и может расширяться уникальным кодом или дополняться какими-то частями из разделяемых файлов.

13.10.1. Макеты

Та информация, которая остается неизменной от страницы к странице, должна содержаться в файлах макета (`Layout`). Если посмотреть на мой сайт **Flenov.info**, то вы увидите шапку (включая большую страницу наверху), рекламу и, конечно, подвал.

Файл шаблона — это такой же файл `*.cshtml`. Ему обычно дают имя `_Layout.cshtml`, а символ подчеркивания в начале имени указывает на то, что это специальный файл, который не вызывался в контроллере и не является результатом работы какого-либо кода. Подчеркивание не обязательно, но для специальных файлов все же желательно его использовать, просто чтобы выделять их визуально.

Остальные страницы сайта могут отображаться внутри основного шаблона. На рис. 13.10 показан сайт, использующий классическую разметку. Прямоугольной рамкой на нем выделен блок, который приходит из конкретной страницы представления, а все остальное — это макет. Впрочем, правая часть приходит на страницу из подключаемого файла.

Итак, с именем файла мы определились, а теперь нам нужно определиться с расположением. До сих пор мы хранили представления в папке с тем же именем, что и у контроллера. Но т. к. файл шаблона является общим для всего сайта, то для таких общих файлов я предпочитаю создавать папку `Views/Shared` (Разделяемые).

Создав такую папку, внутри нее создаем файл макета, щелкнув правой кнопкой мыши на папке **Shared** и выбрав **File | New File**. Тут снова можно выбрать совер-

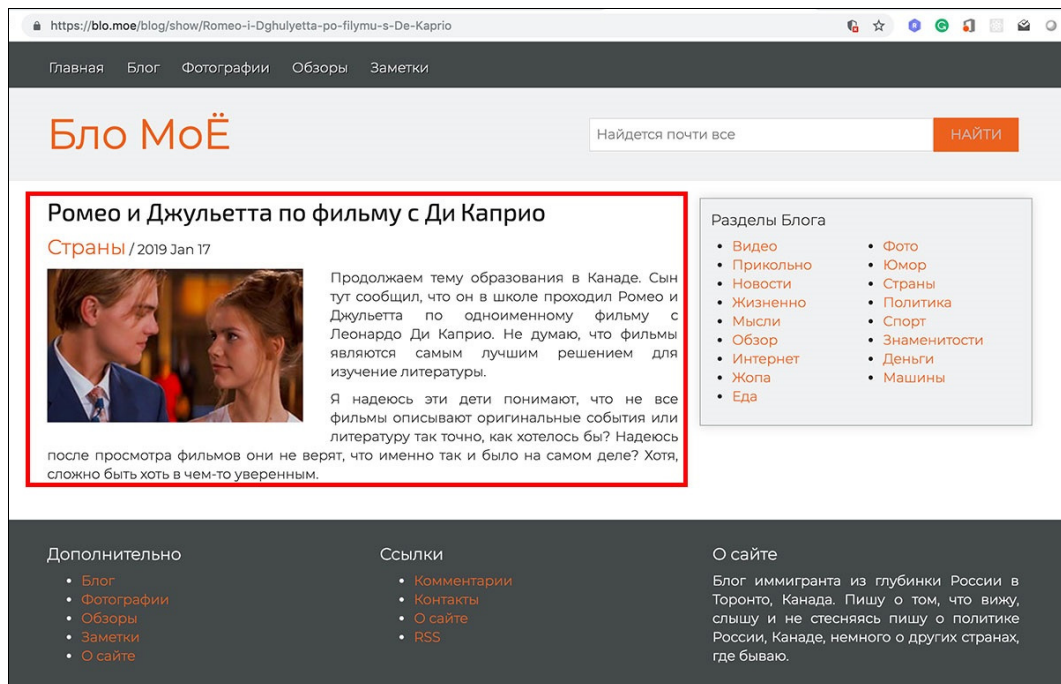


Рис. 13.10. Примерный макет сайта

шенно любой тип файла и ввести в качестве его имени `_Layout.cshtml`, а можно выбрать уже заранее подготовленный шаблон, который просто наполнит файл базовым смыслом. Имя такого шаблона — **MVC View Layout Page** (он может еще называться **Razor Layout** — в зависимости от версии Visual Studio).

Разработчики Visual Studio иногда меняют шаблоны, поэтому вы можете увидеть и другой результат, но в моем случае получился такой код:

```
<!DOCTYPE html>

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>@ViewBag.Title</title>
</head>
<body>
  <div>
    @RenderBody()
  </div>
</body>
</html>
```

Расширение имени файла (`cshtml`) снова указывает на то, что это смесь кода CS и HTML, и здесь мы можем писать абсолютно все то же самое. Просто модели как таковой здесь нет. Зато уже есть два вызова CS-кода:

- ❑ `@RenderBody()` — в этом месте будет отображено содержимое представления, которое мы вызовем из контроллера. То есть если мы скажем, что представление `Rezortest/Index.cshtml` должно использовать этот макет, то содержимое файла `Index.cshtml` будет размещено как раз в месте вызова кода `RenderBody()`;
- ❑ `@ViewBag.Title` — указывает на то, что мы хотим отобразить свойство `Title` от переменной `ViewBag`.

Мы с `ViewBag` еще не знакомились, поэтому сейчас самое время сделать это. Переменная `ViewBag` — динамическая (*dynamic*), и для создания нового поля достаточно просто написать его имя и присвоить значение. Например, так мы можем создать поле `Test`:

```
ViewBag.Test = "Это строка";
```

Переменная `ViewBag` доступна как из контроллеров, так и из представлений, и может использоваться как раз для того, чтобы передавать информацию из контроллеров в представления.

«Стоп! — скажет внимательный читатель. — Вы же утверждали, что для передачи информации из контроллера в представления нужно использовать классы-модели?» Да, именно так. Если вам нужно передать информацию, ваши помыслы должны быть в первую очередь направлены именно на модель. Только в крайних случаях, когда другого выхода нет, можно использовать `ViewBag` и заголовок. А наш случай как раз и может быть таким крайним, потому что в макете у нас нет и не может быть модели. Макет не знает, какую модель отображает текущая страница, потому что он может использоваться с любой страницей, вот поэтому `ViewBag` и становится приемлемым решением.

Я не могу сказать, что в использовании переменной `ViewBag` есть что-то криминальное. Нет, она вполне легальна, но лучше все же стараться по мере возможности обходить ее стороной. На мой взгляд, тогда код будет даже чище.

Вернемся к нашему шаблону. Чтобы он был более наглядным, я добавил к нему заголовок и подвал:

```
<!DOCTYPE html>

<html>
  <head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
  </head>
  <body>
    <div style="background:#eee; font-size:200%; padding:30px">
      Logo
    </div>
    <div>
      @RenderBody()
    </div>
```

```
<div style="background:#eee; padding:30px">
    Copyright: www.flenov.info
</div>
</body>
</html>
```

Если вы работали с CSS и HTML, то, наверное, сейчас упрекнете меня за этот код. Я знаю, что устанавливать стили через атрибут `style` для каждого элемента в отдельности не очень эффективно, но это всего лишь пример, и мы пока не будем усложнять себе жизнь отдельным CSS-файлом, а расположим все форматирование в одном месте, чтобы вы могли тут же увидеть, что я изменил. В реальных приложениях, конечно же, стили будут отдельно: HTML — сам по себе, котлеты — в холодильнике ;-).

Теперь откройте файл `Views\Razortest\Index.cshtml`, и если вы добавляли в него теги типа `html`, `body`, `head`, то удалите их, а оставьте только самое необходимое, — что мы хотим отобразить в теле страницы внутри макета. А в самое начало добавьте такой код:

```
@{
    ViewBag.Title = "This is my index page";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Здесь в первой строке мы устанавливаем заголовок страницы, присваивая строковое значение свойству `ViewBag.Title`. Как мы уже выяснили, макет потом сможет прочитать эту строку. Значения `ViewBag` можно задавать как в CSHTML-файлах, так и в контроллерах. Контроллеры будут более подходящим способом, потому что одно и то же представление может отображаться в ответ на различные действия. Например, форма блога может отображаться как при создании новой записи, так и при редактировании старой, поэтому заголовок страницы должен быть соответствующим. Представление будет одно, а в контроллере методы, скорее всего, будут разными, потому что действия разные, вот поэтому заголовок я чаще всего задаю именно в контроллере.

Во второй строке устанавливается файл макета. У вас на сайте может использоваться несколько различных макетов: один — для основных страниц, другой — для страниц администрирования сайтом, третий — для какого-либо специфичного раздела и т. д. Пока же мы установим макет прямо в файле `Index.cshtml`, но чуть позже познакомимся с тем, как указать, какой макет использовать по умолчанию, и тогда эта строка станет ненужной.

Если запустить этот сайт и перейти на страницу <https://localhost:7261/Razortest/index>, то в результате в браузере мы должны увидеть что-то похожее на рис. 3.11.

Заголовок и подвал отображаются благодаря используемому макету, а центральная часть уникальна для этой страницы.

Теперь можно обновить все представления, чтобы они использовали этот шаблон, и получить преимущество от того, что весь однородный код находится только

в одном месте. Стоит поменять файл макета, и изменения станут тут же видны на всех страницах, которые используют этот макет.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\layout` сопровождающего книгу электронного архива (см. *приложение*).

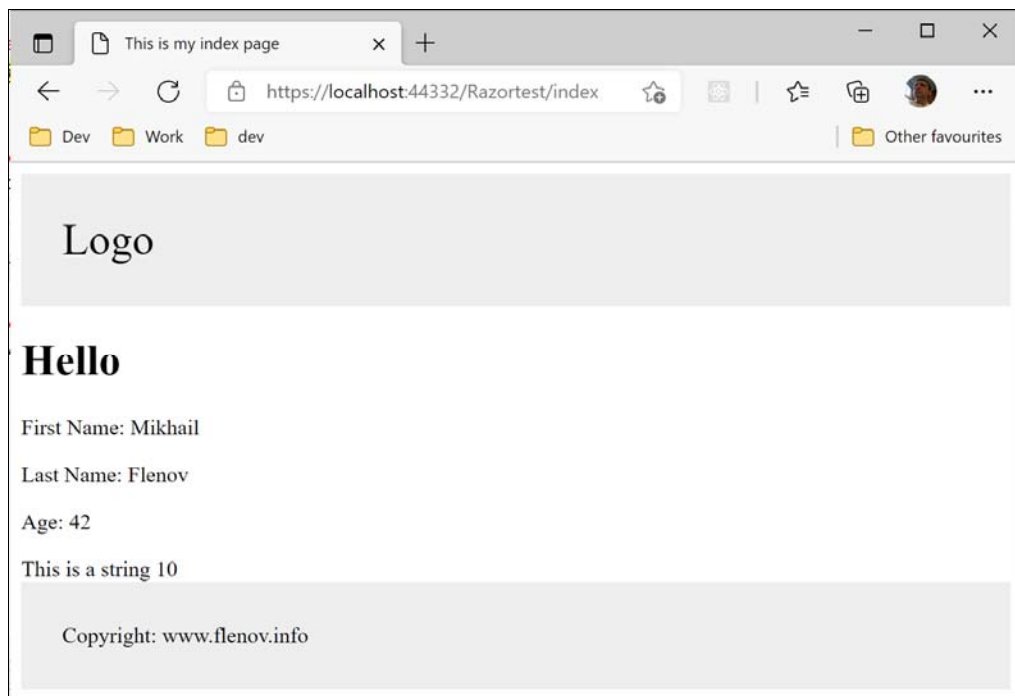


Рис. 13.11. Страница с макетом

13.10.2. Стартовый файл

В предыдущем разделе мы познакомились с тем, как можно создать файл шаблона и использовать его в представлениях. Единственным неудобством при этом является необходимость подключать шаблон в каждом файле представления.

Этого можно избежать, потому что перед тем, как выполнить CSHTML-файл, фреймворк ищет в папке файл `_ViewStart.cshtml` и выполняет его код. Если в текущей папке он не найден, то его ищут в папке уровнем выше.

Давайте создадим такой файл в папке `Views`. Щелкните правой кнопкой мыши на папке **Views** и выберите создание нового файла. В качестве шаблона можно выбрать **RazorPage**, главное — дать файлу имя `_ViewStart.cshtml`.

Откройте созданный файл и удалите все его содержимое, которое сгенерировал в нем мастер создания файлов, а вместо него впишите строку подключения шаблона. То есть файл должен выглядеть так:


```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

В файле `Index.cshtml` строка подключения шаблона уже больше не нужна, и ее отсюда уберите.

Теперь абсолютно все файлы представлений, которые мы будем отображать, по умолчанию станут использовать шаблон `_Layout.cshtml`. Попробуйте запустить приложение и убедиться в этом.

Обратите внимание: я говорю «по умолчанию», потому что мы создали файл `_ViewStart.cshtml` в самом «корне» пути — в папке `Views`. Если создать такой же файл в папке `Razortest` и в нем прописать другой шаблон — например, `_CoolLayout.cshtml`, то все представления из папки `Razortest` будут использовать именно `_CoolLayout`.

В самом представлении тоже можно указать другой шаблон. То есть по умолчанию имя шаблона будет браться из `_ViewStart.cshtml`, а в представлении вы можете указать другой шаблон.

А что если во всех представлениях определенной папки вам нужен общий шаблон, а в одном представлении из них — нет, потому что там будет что-то уникальное? В этом случае шаблон в этом единственном представлении можно обнулить:

```
@{
    Layout = null;
}
```

Дело в том, что в файле представления вы можете указывать шаблоны, как мы это делали в предыдущем разделе, и это указание имеет самый высокий приоритет, потому что вы как бы перезаписываете значение, которое было прописано в файле `_ViewStart.cshtml`. То есть если вы хотите использовать в одном-единственном представлении какой-то уникальный шаблон, то его можно указать непосредственно в `CSHTML`-файле. Например, файл `Views\Razortest\Index.cshtml` может выглядеть так:

```
@model MyWebSite.Model.Person

@{
    ViewBag.Title = "This is my index page";
    Layout = "~/Views/Shared/_Layout1.cshtml";
}

. . .

. . .
```

За счет того, что у нас шаблон по умолчанию указан в самом «корне» пути (в папке `Views`) и все страницы по умолчанию отображаются с использованием этого шаблона, возникает вопрос: а как отобразить представление без шаблона? Его можно, как это показывалось чуть ранее, просто обнулить в файле представления, присвоив `Layout` значение `null`. При этом файл `Views\Razortest\Index.cshtml` может выглядеть так:

```
@model MyWebSite.Model.Person

@{
    ViewBag.Title = "This is my index page";
    Layout = null;
}

. . .
. . .
```

В папке электронного архива для этого раздела можно найти пример, для которого в «корневую» папку представления View я добавил файл `_ViewStart.cshtml`. Этот файл подключает шаблон `_Layout.cshtml`. Я также добавил в папку View папку About (и соответствующий контроллер в папку Controllers), причем содержащийся внутри папки About файл `_ViewStart.cshtml` изменяет шаблон `_Layout.cshtml` на `_CoolLayout.cshtml`. Структуру файлов проекта можно увидеть на рис. 13.12.

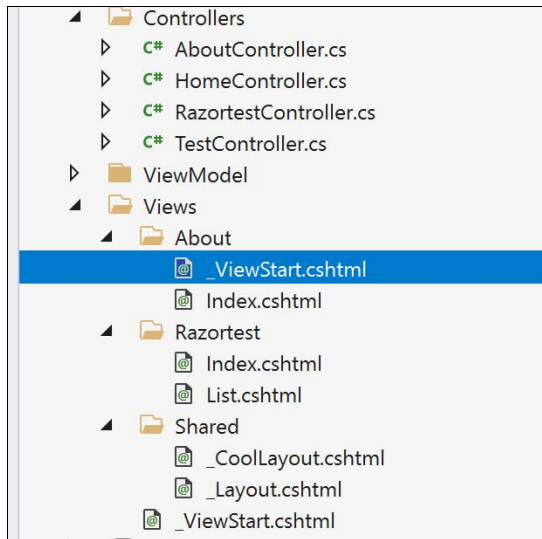


Рис. 13.12. Структура файлов проекта

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\ViewStart` сопровождающего книгу электронного архива (см. приложение).

13.10.3. Встраиваемые представления

Я долго ломал голову над тем, как лучше перевести выражение Partial View. Если переводить его дословно, то это «частичные представления». Попытавшись придать переводу чуть большую осмысленность, учитывая функциональность понятия, я получил вариант «маленькие представления». Суть Partial View — это возможность отрисовать часть страницы. У меня прямо ступор был минут на 10, потому что я не мог начать работать над этой главой и все думал, как же ее назвать.

Поломав немного голову, я решил не переводить выражение `Partial View` дословно и назвать главу по наиболее близкому к его функциональности смыслу — встраиваемые представления.

В MVC мы уже знаем, что есть такое понятие — шаблон. Внутри шаблона мы можем вызвать метод `RenderBody()`, чтобы отрисовать текущую страницу. Вызов `RenderBody()` может быть только один.

А что если на странице будет еще какой-то повторяющийся элемент? Например, у меня на сайте **flenov.info** в разделе блога, в разделе статей и в книгах можно оставлять комментарии. Это совершенно разные страницы, но они используют одну и ту же функциональность — отображение страницы и отображение формы для сохранения комментариев. Это как бы область внутри страницы, которую желательно было бы реализовать отдельно и потом просто вставлять в нужные страницы.

Вот именно это и делают встраиваемые представления (`Partial View`). Они не предназначены для того, чтобы отрисовывать все содержимое страницы. Вместо этого такие представления отвечают за отображение и логику какой-то части страницы, и вы это частичное представление встраиваете куда угодно. Получается, что перевод «частичное представление» тоже верный, и можно было бы привести его здесь в дословном варианте? Да! Но мне все же «встраиваемое представление» почему-то нравится больше. Моя книга — что хочу, то и делаю, поэтому я буду далее использовать это название.

Итак, встраиваемые представления — это такие же представления, как и у страниц, но только когда фреймворк их рендерит, он не задействует шаблон, потому что шаблон для этого не нужен. Шаблон нужен странице, а не отдельной ее части.

Как мы уже определились, представления наши встраиваемые, и встраиваться они могут как в файл шаблона, так и в файл представления страницы. Да, даже в файл другого встраиваемого представления. В любой файл `*.cshtml` (и встраиваемые представления тоже будут иметь такое же расширение) вы можете встроить другое представление. И делается это командой:

```
@Html.Partial("ИмяФайла", Модель)
```

Еще один вариант использования встраиваемых представлений — отображение элементов списка. Чтобы продемонстрировать такой вариант, вернемся к моему сайту. В разделах **Блог** можно увидеть мои постраничные заметки в 10 элементов. Каждая заметка — это отдельное встраиваемое представление, поэтому если бы сайт был написан на C#, то код страницы блога мог бы выглядеть так:

```
foreach (var blogItem in blogItems) {  
    @Html.Partial("blog.cshtml", blogitem)  
}
```

Чтобы работать со встраиваемым представлением, сначала создадим его файл. Файл встраиваемого представления можно поместить в папку `Shared` или в ту же папку, из которой мы будем рисовать этот файл. Я иногда помещаю такие файлы в папку `Widget` (Виджет). Но сейчас я помешу его в папку `Views\RazorTest`. Для этого щелкаем правой кнопкой мыши на этой папке и создаем новый файл. Раньше

в Visual Studio, кажется, был для таких файлов специальный шаблон, но в версии для Mac я его не увидел. Впрочем, это всего лишь шаблоны, и мы можем выбрать тут совершенно любой тип файла, главное — дать ему правильное расширение и правильный контент.

Для Partial View рекомендуется ставить в начале имени префикс в виде символа подчеркивания. Это не обязательно, а только рекомендация, чтобы визуально отличать файлы полноценных представлений от встраиваемых или каких-либо системных, как мы уже делали с файлом `_Layout.cshtml`.

Так что в качестве имени файла я принял `_Item.cshtml`, и Visual Studio создал для нас файл, содержимое которого можно очистить. Вместо него напишите следующий код:

```
@model MyWebSite.ViewModel.Person
<li>
    FirstName: @Model.FirstName
    Last Name: @Model.LastName
</li>
```

Если вы внимательно читали предыдущие главы, то в этом файле для вас нет совершенно ничего нового. В самом начале мы задаем тип модели, который ожидаем во время вызова представления, а потом идет смесь кода HTML и C#, в которой я просто отображаю имя и фамилию.

В результате мы обзавелись встраиваемым представлением, которое можем куда-нибудь внедрить, так давайте это и сделаем. В предыдущих примерах у меня присутствовал контроллер `RazortestController` с методом `List`, который отображал список людей. Мы использовали его, когда я показывал различные варианты циклов в представлениях. Давайте перепишем тот пример с использованием встраиваемого представления и отобразим людей в списке с его помощью. Контроллер остается тем же — мы меняем только отображение, поэтому открываем файл `List.cshtml` и все его содержимое заменяем на следующее:

```
@foreach (var p in Model)
{
    @Html.Partial("_Item", p);
}
```

Теперь весь код сводится к циклу, который перебирает всех людей из модели и для каждого человека вызывает отображение встраиваемого представления, — мы встраиваем его в текущую позицию с помощью следующего кода:

```
@Html.Partial("_Item", p);
```

Первый параметр — это имя файла, а второй — модель типа `Person`. Именно ее и ожидает наше представление.

Преимущество встраиваемого представления заключается в том, что теперь `_Item` работает как виджет. Мы можем отображать его на любой странице сайта и вызывать его откуда угодно. И если в определенный момент захочется поменять внеш-

ний вид этого виджета (карточки человека), то достаточно поменять ее в одном файле `_Item.cshtml`, и изменения вступят в силу везде.

Кстати, использование метода `Partial` — это несколько устаревший подход, и сейчас MS рекомендует использовать асинхронную версию `PartialAsync`, которая защищена от блокировок:

```
@await Html.PartialAsync("_Item", p)
```

Метод `Partial` тоже будет работать, просто во время компиляции вам покажут предупреждение с предложением использовать асинхронную версию.

Есть еще пара методов: `RenderPartial` и `RenderPartialAsync`. Различия этих двух методов в том, что они вызывают отрисовку представления, но при этом не возвращают его в качестве результата, а отображают его.

Посмотрим на следующие два примера:

```
@await Html.PartialAsync("_Item",
    new MyWebSite.Model.Person() {FirstName="Михаил", LastName="Фленов"})
@{
    await Html.RenderPartialAsync("_Item",
        new MyWebSite.Model.Person()
        {
            FirstName = "Михаил",
            LastName = "Фленов"});
}
```

В первом случае вызывается метод `PartialAsync`, который возвращает результат отрисовки, и мы его отображаем за счет того, что ставим впереди символ `@`. Точно так же мы отображаем и имя, когда пишем `@Model.FirstName`. Одиночный символ `@` как раз и служит для отображения.

Второй метод: `RenderPartialAsync` — ничего не возвращает, поэтому нельзя написать:

```
@await Html.RenderPartialAsync(. . . .
```

Такой вариант закончится ошибкой. Вместо этого я перехожу в режим кода `@{`, а уже потом вызываю `RenderPartialAsync`, который ничего не вернет, но зато все отобразит.

Как можно использовать эти различия? Допустим, вы собираетесь вызвать представление, но хотите не отображать его в текущей позиции, а сохранить результат, а потом еще и обработать. Такое возможно:

```
@{
    var result = Html.Partial("_Item",
        new MyWebSite.Model.Person()
        {
            FirstName = "Михаил",
            LastName = "Фленов"
        });
}
```

Здесь мы не отображаем представление, а вызываем его, и результат рендеринга сохраняем в переменной `result`. Теперь эту переменную можно как-то еще обработать, передать кому-нибудь или даже просто отобразить, если написать: `@result`.

А если встраиваемое представление только отображает информацию и вам не нужна модель, то ее можно и не указывать, — здесь может быть просто HTML-код, который подключается из разных мест.

На всех моих сайтах я обязательно использую следующую запись:

```
@Html.Partial("_ads.cshtml")
```

Модель здесь не передается, потому что в файле `_ads.cshtml` нет каких-то данных, там присутствует фиксированный HTML-код, который генерирует Google, чтобы я мог отображать на своем сайте рекламу. И если Google когда-либо поменяет код, который должен быть на моем сайте, мне достаточно будет поменять только один файл.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Partial` сопровождающего книгу электронного архива (см. *приложение*).

13.10.4. Компоненты

Представления, которые мы рассматривали в предыдущем разделе, отлично работают, если у вас есть под рукой готовая модель, которая может отобразить себя, но это далеко не всегда возможно. Например, у меня на сайте в правом верхнем углу отображается имя пользователя, если он зарегистрирован. Этот кусочек страницы строится в шаблоне страницы. Неужели мне нужно передавать шаблону `_Layout.cshtml` какую-то модель? Это вообще возможно?

Проблему с выводом имени текущего пользователя в файле шаблона можно решить двумя методами:

- ☐ создать где-то в модели статичный метод и обратиться к нему;
- ☐ использовать компонент.

Я предпочитаю второй метод, и он как раз отлично подходит для нашего случая.

Компоненты по своему смыслу очень похожи на встраиваемые представления — это такие же CSHTML-файлы, при отображении которых не будет задействован шаблон.

Разница только в том, откуда берется модель для файла представления. В случае с представлением модель нужно передать ему в качестве параметра при вызове, потому что когда мы отображаем встраиваемое представление, фреймворк сразу же идет и ищет именно файл представления. В случае с компонентами к представлению привязан C#-код, в котором вы можете сформировать модель и передать ее представлению.

Получается, что компонент — это как бы независимый элемент MVC. Вы говорите: отобразите мне в этой части страницы компонент `Login`, и фреймворк находит сна-

чала код компонента, который формирует модель, а потом вызывает представление для отображения содержимого модели.

Убедимся на практике, что это действительно так прекрасно.

Код компонентов хранится в отдельной папке с названием **ViewComponents**. У вас ее, скорее всего, нет, поэтому придется создать ее так же, как мы создавали папку для контроллеров. Щелкаем правой кнопкой мыши на имени проекта и выбираем **Add | New Folder**. Затем внутри папки **ViewComponents** создаем файл — назовем его **UserViewComponent.cs**, — в котором и будет прописан код. Именование очень важно, потому что если мы просим фреймворк отобразить компонент **User**, то он будет искать класс с именем **UserViewComponent**.

Далее:

1. Надо добавить в этот файл пространство имен **Microsoft.AspNetCore.Mvc**, потому что мы будем работать с компонентом.
2. Класс должен наследовать **ViewComponent**, поэтому обязательно добавим и это наследование.
3. Нужно реализовать метод **Invoke**, который должен возвращать тип **IViewComponentResult**.

Метод **Invoke** работает так же, как и методы в контроллерах. Только если в контроллерах один класс может обрабатывать несколько различных URL, то здесь это один класс на один компонент, поэтому нужен один метод. Полный код моего примера будет выглядеть так:

```
using Microsoft.AspNetCore.Mvc;
using MyWebSite.ViewModel;

namespace MyWebSite.ViewComponents
{
    public class UserViewComponent : ViewComponent
    {
        public IViewComponentResult Invoke()
        {
            var model = new UserModel()
            {
                IsLoggedIn = true,
                UserName = "mflenov"
            };
            return View("User", model);
        }
    }
}
```

В методе **Invoke** я создаю экземпляр модели **UserModel** с парочкой параметров и имитирую, как будто пользователь уже вошел в систему, и его имя **mflenov**. В реальном приложении тут может также присутствовать проверка сессии на предмет наличия индикатора, что пользователь вошел в систему.

UserModel — это класс, который еще нужно создать. Добавьте в папку ViewModel новый файл с именем UserModel.cs и в него поместите следующий код:

```
namespace MyWebSite.ViewModel
{
    public class UserModel
    {
        public bool IsLoggedIn { get; set; }
        public string UserName { get; set; }
    }
}
```

Результат метода возвращается точно так же, как и в случае методов контроллера — вызывается метод View, которому передаются два параметра: имя представления (CSHTML-файла, в котором будет находиться HTML-компонент) и модель.

Представления компонентов могут располагаться в разных местах, и чтобы узнать их место, достаточно встроить вызов компонента (об этом чуть далее) в страницу и запустить приложение. Когда фреймворк .NET не сможет найти CSHTML-файл, он завершит работу с сообщением об ошибке, и в этом сообщении будут указаны пути, где он пытался найти представление. Разбор этого вы можете оставить себе в качестве домашнего задания, а я рекомендую располагать компоненты по следующему пути: Views\Shared\Components\ИмяКомпонента\ИмяКомпонента.cshtml.

В результате файлы будут хорошо сгруппированы по папочкам, и фреймворк сумеет работать с таким шаблоном. В нашем случае компонент называется User, а значит, путь будет такой: Views\Shared\Components\User\User.cshtml.

В представлении компонента можно писать такой же код, который мы писали в любом другом представлении. Для нашего примера я написал следующий:

```
@model MyWebSite.ViewModel.UserModel

@if (Model.IsLoggedIn)
{
    <text>Welcome: @Model.UserName</text>
}
else
{
    <text>Please login</text>
}
```

Вот теперь компонент можно считать законченным, и осталось только отобразить его на странице. Открываем шаблон _Layout и добавляем в него:

```
@await Component.InvokeAsync("User")
```

Таким образом, в месте, где мы хотим отобразить компонент, просто вызываем метод Component.InvokeAsync, а в качестве параметра указываем имя компонента. Поскольку в названии присутствует слово Async, значит, метод асинхронный, и нам нужно указать еще и await.

Можно запустить приложение и убедиться, что оно работает.

Есть еще один способ вызова компонента — он считается более современным, потому что задействует теги, и я бы рекомендовал использовать именно его:

```
<vc:user></vc:user>
```

Тег `<vc:>`, видимо, означает `ViewComponent`. А после двоеточия указываем, какой именно компонент мы хотим отобразить.

Но если сейчас запустить сайт, то работать это не будет, потому что по умолчанию фреймворк может отображать только теги, о которых он знает. Вариант с `Component.InvokeAsync` будет работать без дополнительных усилий, а вот для тега нужен дополнительный шаг. Надо показать ему, что у нас есть свои компоненты, и для этого создайте в папке `Views` файл `_ViewImports.cshtml`. Мы уже знакомы с файлом `_ViewStart.cshtml`, в котором мы можем указать имя шаблона, а `_ViewImports.cshtml` — отличное место для того, чтобы указывать, какие пространства имен мы хотим подключить в свои представления, да и описание тегов также будет круто прописать именно здесь. Итак, мы создали файл с таким именем и помещаем в него всего одну строку:

```
@addTagHelper *, MyWebSite
```

В нашем случае `MyWebSite` — это пространство имен моего приложения, и если вы посмотрите класс компонента, то он был именно в этом пространстве.

Все, можно запускать сайт, и теперь он должен работать и с тегами.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter13\Component* сопровождающего книгу электронного архива (см. приложение).

13.10.5. Секции

Когда мы отображаем страницу, то все данные будут отображаться в месте, где в файле шаблона находится вызов `@RenderBody()`, и обычно это где-то в середине. А что если мы хотим добавить на страницу какие-нибудь файлы JavaScript-скриптов или изменить что-либо в заголовке страницы, в секции `<head>`. Компоненты не помогут, нужно реально иметь возможность вывода сразу в несколько мест на странице.

На помощь нам придет элемент `@section`. В файле шаблона с его помощью мы можем указать секции, в которые могут выводить информацию представления.

Популярным решением является секция скриптов, которая обычно находится внизу страницы, — давайте добавим ее в файл шаблона прямо перед закрывающимся тегом `</body>`:

```
<!DOCTYPE html>
<html>
<body>
  <div style="background:#eee; font-size:200%; padding:30px">
    Logo
```

```

    @RenderSection("Head", required: false)
</div>
<div>
    @RenderBody()
</div>
<div style="background:#eee; padding:30px">
    Copyright: www.flenov.info
</div>

@RenderSection("Scripts", required: false)
</body>
</html>

```

У команды `RenderSection` есть два параметра: имя секции и указание на то, является ли секция обязательной.

На странице может быть больше одной секции, каждая из них должна быть уникальной, и имя как раз задает эту уникальность. В приведенном примере создаются одна секция — `Head` и еще одна секция — `Scripts`.

Обе секции обозначены необязательными (`false`), а это значит, что представления не обязаны выводить в каждую из них какие-либо данные. Если секцию сделать обязательной, то представление обязано будет предоставить контент для секции, иначе произойдет ошибка.

Теперь посмотрим, как мы можем предоставить содержимое для этих секций в представлении на примере файла `Razertest\Index.cshtml`:

```

@section Scripts {
    <script type="text/javascript" src="~/scripts/cart.js"></script>
}

@section Head {
    <p>Этот текст отрисован в секции из представления</p>
}

```

После команды `@section` идет имя секции, контент которой мы хотим предоставить, а в фигурных скобках после этого записан уже сам контент.

Таким образом представление может одновременно выводить контент сразу в несколько мест в шаблоне.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Section` сопровождающего книгу электронного архива (см. *приложение*).

13.11. Работа с формами

До сих пор я старался использовать максимально простой код без украшений, но для работы с формами все же придется работать с более сложной разметкой. Если вы запустите код примера из соответствующей папки сопровождающего книгу электронного архива, то, скорее всего, увидите совершенно обновленную страницу.

Первым делом я подключил поддержку статичных файлов в файле Startup.cs:

```
app.UseStaticFiles();
```

Если файла Startup.cs в вашей конфигурации нет, значит, вы выбрали новый шаблон, и эту запись нужно будет поместить в файл Program.cs.

Это необходимо, потому что я буду использовать отдельные CSS-файлы, и без подключения статичных файлов браузер не сможет загрузить такой файл.

В раздел <head> файла шаблона _Layout.cshtml я добавил подключение CSS-файла:

```
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="/css/stylesheet.css"/>
</head>
```

Сам файл я приводить не буду, вы его без проблем сможете найти в папке Source\Chapter13\Forms\wwwroot\css\ сопровождающего книгу электронного архива. Описание стилей выходит за рамки этой книги.

В раздел <header> шаблона я добавил меню, которое позволит вам путешествовать по различным разделам примеров, которые были созданы в этой главе:

```
<header>
    <span class="menu">
        <a href="/Razortest/Index">Razer</a>
        <a href="/Razortest/List">Списки</a>
        <a href="/Login">Формы</a>
    </span>
</header>
```

Ну и последнее — я изменил метод Index в файле HomeController.cs, чтобы он возвращал представление. До сих пор он возвращал строку, поэтому шаблон на главной странице не работал. Теперь сайт выглядит чуть более элегантно — как показано на рис. 13.13 (да и сам этот номер элегантен, не правда ли?).

Теперь начнем непосредственно знакомиться с формами и сделаем это на примере формы для входа на сайт. Для входа на сайт обычно используют адрес E-mail и пароль. Эти данные где-то нужно хранить, и в .NET в таком случае надо создавать модель. Модель нужна представлению, а это значит, что нужна модель представления.

Итак, создаем в папке ViewModel новый файл с именем LoginModel.cs и в классе этого файла описываем нужные нам данные:

```
namespace MyWebSite.ViewModel
{
    public class LoginModel
    {
        public string Email { get; set; }
```

```

        public string Password { get; set; }
    }
}

```

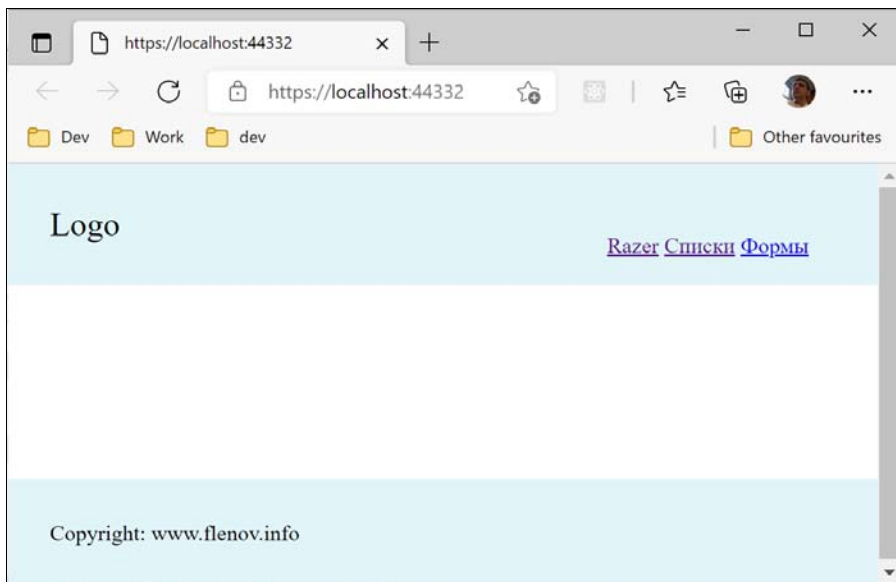


Рис. 13.13. Улучшенный вид сайта

Затем создаем файл контроллера LoginController.cs со следующим содержимым:

```

using Microsoft.AspNetCore.Mvc;
using MyWebSite.ViewModel;

namespace MyWebSite.Controllers
{
    public class LoginController : Controller
    {
        [HttpGet]
        public IActionResult Index()
        {
            ViewBag.Title = "GET-запрос";
            return View(new LoginModel());
        }
        [HttpPost]
        public IActionResult Index(LoginModel model)
        {
            ViewBag.Title = "POST-запрос";
            return View(model);
        }
    }
}

```

Пока контроллер очень простой — здесь два метода `Index` с разными параметрами:

- ❑ первый метод будет вызываться, когда пользователь просто загрузит страницу. Если пользователь в первый раз обращается к `\Login\Index`, то никакая модель еще не нужна, поэтому в параметрах ничего нет. Зато при вызове представления мы будем передавать ему пустую новую модель `LoginModel`. Зачем это нужно?

Допустим, что пользователь отправляет форму на сервер, и произошла ошибка. Очень часто форма отображает все те же данные и просит пользователя исправить ошибку и повторить попытку. Для этого представление должно иметь возможность отобразить данные из модели. Можно создать два представления: одно — с пустыми данными для первой загрузки, а второе — предназначенное получать модель с данными для отображения. А можно иметь только одно представление, просто в изначальном варианте будет содержаться пустая модель;

- ❑ второй метод `Index` будет вызываться, когда пользователь уже ввел данные и направил их на сервер с помощью метода `HttpPost`. В качестве параметра второй метод получает модель `LoginModel` с данными от пользователя, и их же мы будем отдавать обратно представлению для отображения.

Чтобы было наглядно, я в контроллере устанавливаю разные значения для заголовка страницы.

Теперь создаем файл представления `Login\index.cshtml`. Текст формы можно реализовать двумя способами: задействовать специальные инструкции платформы или использовать чистый HTML. Посмотрим сначала на вариант, который задействует специальные инструкции:

```
@model MyWebSite.ViewModel.LoginModel

@using (Html.BeginForm())
{
    <label>E-mail</label>
    @Html.TextBoxFor(m => @Model.Email)
    <br />
    <label>Password</label>
    @Html.PasswordFor(m => @Model.Password)
    <hr />
    <button>Войти</button>
}
```

Вначале мы подключаем модель, а потом используем метод `Html.BeginForm` для создания формы. Когда мы вызываем этот метод, то на самом деле на странице просто создается тег `<form>`.

Затем создаем блок для ввода E-mail:

```
@Html.TextBoxFor(m => @Model.Email)
```

В этот момент на страницу будет выведен тег `<input>` с именем `Email` и значением, которое находится в модели `Model.Email`:

```
<input type="text" name="Email" value="" />
```

Это все прекрасно работает для тех, кто предпочитает синтаксис C#. Но я не использую этот подход по двум причинам: я привык к чистому языку разметки HTML, и чистый HTML проще поддерживать в команде, где есть веб-программисты.

В больших командах за верстку (создание HTML-кода) отвечают веб-программисты. Они получают от дизайнеров макеты страниц и превращают их в HTML-код. Если у вас в команде применяется такое разделение труда, то, скорее всего, кто-то из веб-программистов передаст вам готовый HTML-код, который вы должны будете использовать в своей форме, и он может выглядеть так:

```
<form method="post" action="/login/index">
  <label>E-mail</label>
  <input type="text" name="Email" value="" />
  <br/>
  <label>Password</label>
  <input type="password" name="Password" value="" />
  <hr />
  <button>Войти</button>
</form>
```

Вы можете потратить свое время и переписать этот код с помощью помощников, которые, по задумке Microsoft, упрощают жизнь, но, на мой взгляд, это только усложняет поддержку. Намного проще максимально использовать HTML и внедрять Razor только там, где это необходимо, а в нашем случае достаточно лишь установить значения у элементов `<input>` и добавить в самом начале тип модели:

```
@model MyWebSite.ViewModel.LoginModel

<form method="post" action="/login/index">
  <label>E-mail</label>
  <input type="text" name="Email" value="@Model.Email" />
  <br/>
  <label>Password</label>
  <input type="password" name="Password" value="@Model.Password" />
  <hr />
  <button>Войти</button>
</form>
```

Здесь я максимально использую HTML-код, который в реальной работе мне мог дать веб-программист или верстальщик. Если возникнет проблема с разметкой, я могу дать этот код верстальщику, который поправит любые проблемы, а я буду заниматься своим делом — программировать.

Значит ли это, что я против помощников? Нет, я не против, и если вам удобно использовать методы `@Html`, то никто возражать не станет. И хотя в этой книге я прибегаю к чистому HTML только из личных предпочтений, тем не менее, как видите, показываю вам оба возможных варианта.

Если вы используете HTML, то очень важно давать правильные имена элементам управления — они должны соответствовать именам полей в модели. Когда пользо-

ватель будет направлять заполненную форму на сервер, то для каждого поля `input` в форме браузер создаст пару из имени и значения и направит на сервер уже эту информацию.

Как можно видеть из приведенного ранее кода, у меня на форме имеются два поля с именами: `Email` и `Password`. При этом на сервере мы ожидаем модель `LoginModel`, у которой тоже есть поля `Email` и `Password`. Так как имена совпадают, то на сервере фреймворк сможет заполнить их значениями.

Попробуем добавить к форме еще одно поле — `username`:

```
<input type="text" name="username" value="@Model.Email" />
```

Но у модели `LoginModel` нет поля с именем `username`, а значит, мы не сможем увидеть его через этот класс. Так что, работая с чистым HTML, единственное, о чем вам придется заботиться, — правильно именовать поля.

Попробуйте запустить приложение и нажать кнопку **Войти**. Очень сложно будет увидеть, что данные реально направились на сервер, и мы видим уже новую форму, просто с теми же параметрами, но небольшой блик прорисовки можно заметить. Вы можете увидеть разницу в заголовке страницы — ведь я даю в контроллере разные имена `ViewBag.Title`. Чтобы было еще нагляднее, этот заголовок можно вывести и в теле страницы:

```
<h1>@ViewBag.Title</h1>
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Forms` сопровождающего книгу электронного архива (см. *приложение*).

13.12. Проверка данных

Когда пользователь отправляет форму на сервер, нам нужно иметь возможность проверить введенные им данные и убедиться, что предоставлена корректная информация. Пока еще мы можем загрузить нашу форму входа на сайт и сразу нажать кнопку **Войти**, даже не предоставляя E-mail и пароль (оставив эти поля пустыми).

Но мы должны иметь возможность проверить данные, и у фреймворка есть достаточно широкие полномочия для автоматической проверки данных, которые можно определить с помощью атрибутов для полей модели.

Для формы входа на сайт оба предлагаемых к заполнению поля должны быть обязательными. Вот сначала и сделаем все поля обязательными, и для этого перед каждым поставим атрибут `[Required]`:

```
public class LoginModel
{
    [Required]
    public string Email { get; set; }
```

```
[Required]
public string Password { get; set; }
}
```

Что изменилось? Если сейчас запустить сайт и попробовать отправить пустую форму, то ничего не изменится. Неужели проверка не работает? Она работает, просто мы этого еще не видим. Нужно как-нибудь добавить отображение ошибок. Для этого в CSHTML-файле надо указать в том месте, где вы хотите увидеть сообщение об ошибке, вызов метода `Html.ValidationMessageFor`. Этому методу нужно указать в качестве параметра имя поля, для которого надо выводить сообщение об ошибке. Для E-mail отображение ошибки будет выглядеть так:

```
@Html.ValidationMessageFor(m => m.Email)
```

Мы можем выводить сообщения об ошибках в нужным местах для каждого поля. А можно еще в определенном месте вывести все сообщения об ошибках сразу:

```
@Html.ValidationSummary()
```

Я обновил свое представление с учетом отображения ошибок следующим образом:

```
@model MyWebSite.ViewModel.LoginModel

<form method="post" action="/login/index">
    <h1>@ViewBag.Title</h1>
    @Html.ValidationSummary()
    <label>E-mail</label>
    <input type="text" name="Email" value="@Model.Email" />
    @Html.ValidationMessageFor(m => m.Email)
    <br />
    <label>Password</label>
    <input type="password" name="Password" value="@Model.Password" />
    @Html.ValidationMessageFor(m => m.Password)
    <hr />
    <button>Войти</button>
</form>
```

Вот теперь, если запустить сайт и попытаться отправить форму без данных, то вы увидите на форме ошибку. Если на форме есть ошибка, то в этом месте будет добавлена HTML-строка (рис. 13.14):

```
<span class="field-validation-error" data-valmsg-for="Email" data-valmsg-replace="true">The Email field is required.</span>
```

Обратите внимание, какой класс у этого тега: `field-validation-error`. Если вы хотите оформить эту ошибку с помощью CSS, то нужно использовать этот класс.

Отлично, теперь мы видим сообщение об ошибке, но она на английском, однако даже в Канаде при работе над сайтом для американского подразделения Sony у меня были проблемы с тем, что клиенту не нравились сообщения об ошибках по умолчанию, и он хотел их поменять. Не вопрос, это делается очень легко — мы можем указать нужное сообщение об ошибке в атрибуте:

```
[Required(ErrorMessage = "Email обязателен.")]
```

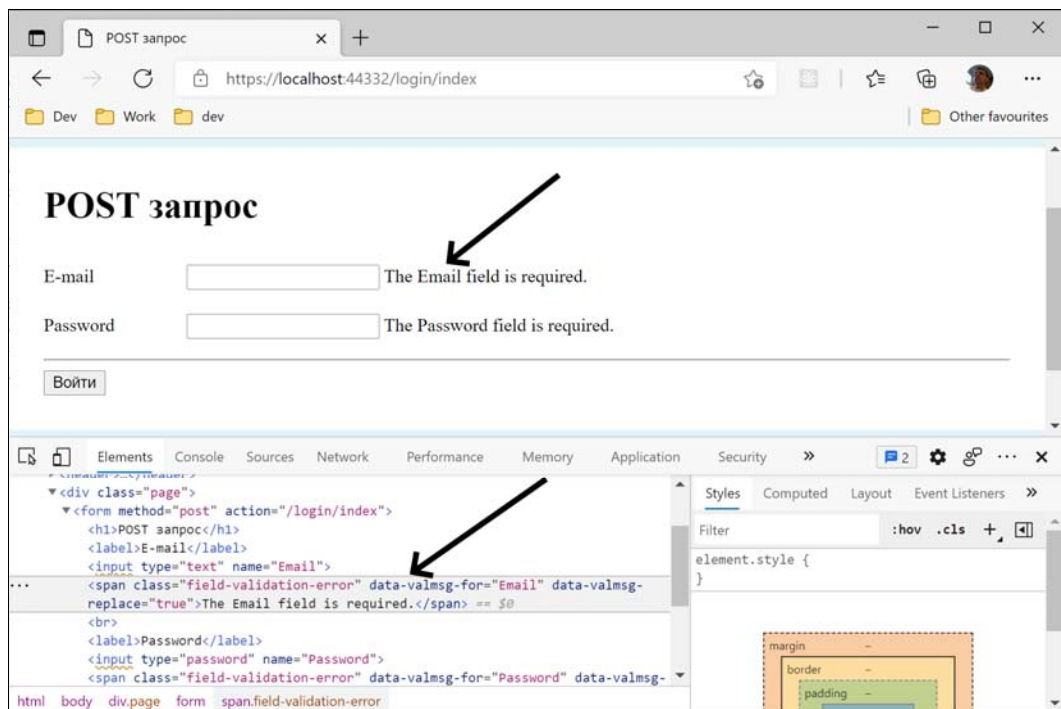



Рис. 13.14. Улучшенный вид сайта

Теперь нам может захотеться ограничить размер вводимых данных, и это можно сделать с помощью атрибута `MaxLength`, которому в скобках надо передать максимальное количество символов:

```
[MaxLength(30)]
```

И можно также указать еще и сообщение об ошибке:

```
[MaxLength(30, ErrorMessage = "Слишком много букв")]
```

В отношении адреса E-mail было бы неплохо еще убедиться, что сам E-mail корректный. Для этого можно использовать регулярные выражения:

```
[RegularExpression(@"^(?!\.)(\"\"([^\"]\\r\\n|\\\"\\\"\\r\\n)\"\"|\"\" +
@\"([a-z0-9!#$%&'*/+=^`{|}~]|(?<!\.\\.)\"\"|(?<!\.\\.)\"\" +
@\"([a-z0-9]\\\\w\\.\\.)\"\"*([a-z0-9]\\\\. [a-z] [a-z\\\\.]\"\"*([a-z] $\"\"))]
```

Теперь мы сделали все основные проверки, и в коде контроллера, наверное, хотим задать логику: если все данные корректные, то выполнить такую-то операцию, если есть ошибки, то отобразить форму, чтобы пользователь исправил ошибку. Но если все проверки происходят автоматически, то как мы можем узнать, были ли ошибки?

Для этого у контроллера есть свойство `ModelState`, которое как раз и отражает состояние модели. Если `ModelState.IsValid` равно `true`, то все проверки прошли успешно.

Следующий пример метода контроллера показывает, как может быть реализована логика:

```
public IActionResult Index(LoginModel model)
{
    if (ModelState.IsValid)
    {
        return Redirect("/");
    }
    ViewBag.Title = "POST запрос";
    return View(model);
}
```

Здесь мы познакомились с новым методом: `Redirect`. Он возвращает `IActionResult`, который вынуждает браузер загрузить страницу, указанную в качестве параметра. В нашем случае в качестве параметра указана домашняя страница — а это значит, что если все указанное пользователем корректно, то мы переадресовываем пользователя на домашнюю страницу.

А что если нужно делать какие-либо проверки, которые невозможно организовать с помощью встроенных атрибутов? Для формы входа на сайт нам, скорее всего, надо бы еще проверить, есть ли пользователь в базе данных и правилен ли его пароль. Так где это делать?

Нестандартные проверки можно сделать прямо в контроллере, но есть способ лучше. Если модель представления реализует интерфейс `IValidatableObject`, то, значит, у класса будет метод `Validate`, фреймворк вызовет его, и мы в нем сможем совершить дополнительные проверки:

```
public class LoginModel:IValidatableObject
{
    // здесь объявление полей Email и Password
    // . . .
    // . . .

    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        if (Email != "test@email.com")
        {
            yield return new ValidationResult("Неизвестный пользователь",
                new string[] { "Email" });
        }
        else if (Password != "pass")
        {
            yield return new ValidationResult("Неверный пароль",
                new string[] { "Password" });
        }
    }
}
```

Метод `Validate` возвращает перечисление из ошибок, которые мы найдем. Сначала проверяется свойство `Email`, и если в нем находится что угодно, но не

test@email.com, то возвращается сообщение об ошибке, что такой E-mail не найден. Поскольку мы еще не умеем работать с базами данных, то — ради примера — я проверяю здесь правильность ввода относительно жестко прописанного в коде адреса.

Для того чтобы вернуть, создается экземпляр класса `ValidationResult`, конструктор которого получает два параметра: сообщение об ошибке и список полей, которые мы посчитали неверными, чтобы ошибка отображалась рядом с полем, которое мы тут указали.

Таким образом в методе `Validate` мы можем реализовывать совершенно любую логику, никаких ограничений нет.

13.13. Работа с сессиями

Прежде чем мы сможем использовать в своем коде сессии, мы должны включить эту функциональность, потому что, как вы уже заметили, в .NET Core/.NET 7 по умолчанию все отключено.

Возможности настраиваются и подключаются в файле `Startup.cs` (если его в вашей конфигурации нет, то в файле `Program.cs`). Здесь в методе `Configure` мы включим сессии, вызвав следующий метод:

```
app.UseSession();
```

Теперь в методе `ConfigureServices` нужно настроить сессию:

```
services.AddSession(options =>
{
    options.Cookie.Name = ".MyWebSite.Session";
    options.IdleTimeout = TimeSpan.FromMinutes(10);
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
});
```

В принципе, все настройки можно оставить по умолчанию, но в реальной жизни иногда приходится настраивать параметры сессии:

- ☐ `options.Cookie.Name` — это имя Cookie-параметра, к которому будет привязана сессия;
- ☐ `options.IdleTimeout` — сколько времени будет жить сессия;
- ☐ `options.Cookie.HttpOnly` — это необходимо, чтобы Cookie был доступен только на сервере и JS доступа к нему не имел;
- ☐ `options.Cookie.IsEssential` — указывает на то, что параметр является важным, и без него работа приложения невозможна.

Я установил время жизни сессии в 10 минут — если пользователь не проявит активность в течение этого времени, сессия будет уничтожена. Для таких сайтов, как социальные сети, можно держать сессию активной достаточно долго. А вот для банков или интернет-магазинов лучше закрывать сессию через какое-то время.

Дело в том, что по умолчанию сессии живут, пока открыт браузер. Стоит закрыть браузер и открыть заново, все сессии будут сброшены. Но в последнее время пользователи не так часто закрывают браузер и могут держать вкладки открытыми очень долго. А в мобильном браузере у меня на планшете есть постоянно открытые вкладки.

Итак, настройка завершена, теперь посмотрим, как можно сохранить в сессии какое-нибудь значение и потом его отобразить.

У нас есть контроллер для входа на сайт, где мы проверяем имя и пароль, и если имя введено верно, то мы как бы авторизуем пользователя. Самый простой способ сохранить факт авторизации — использовать сессию, сохранив в ней какой-либо маркер, который будет говорить, что пользователь авторизован.

Для работы с сессией из контроллера есть свойство `HttpContext.Session`, но у него очень мало методов, и они неудобные. Если же мы подключим в файле пространства имен `Microsoft.AspNetCore.Http`:

```
using Microsoft.AspNetCore.Http;
```

то у объекта сессии появятся удобные для использования методы: `GetString` и `SetString`. Если модель корректная, то пользователь авторизовался, и мы можем добавить в сессию параметр с помощью метода `SetString`:

```
public IActionResult Index(LoginModel model)
{
    if (ModelState.IsValid)
    {
        HttpContext.Session.SetString("username", model.Email);
        return Redirect("/");
    }
    ViewBag.Title = "POST запрос";
    return View(model);
}
```

У нас есть возможность создавать несколько различных сессий с различными именами, и первый параметр у метода `SetString` — это как раз имя. Второй параметр — значение.

Сохранив E-mail, мы его сможем теперь отобразить на форме в шаблоне. А для этого использовать компоненты, которые рассматривали в *разд. 13.10.4*. Я просто скопировал сюда тот код и немного подправил код компонента:

```
public IViewComponentResult Invoke()
{
    string username = HttpContext.Session.GetString("username");
    var model = new UserModel()
    {
        IsLoggedIn = username != null,
        UserName = username
    };
}
```

```
return View("User", model);  
}
```

Здесь я пытаюсь прочитать значение сессии с именем `username` с помощью метода `HttpContext.Session.GetString`. Если у вас произойдет ошибка компиляции на этом методе, то вы, скорее всего, забыли подключить пространство имен `Microsoft.AspNetCore.Http`.

Попробуйте сейчас запустить сайт, вызвать форму для входа и после входа открыть утилиты разработчика. В браузере Chrome для этого нажмите клавишу `<F12>` или комбинацию клавиш `<Ctrl>+<Shift>+<I>`, и должна открыться панель с различными вкладками. Перейдите на вкладку **Application** (Приложение), с левой стороны в разделе **Storage** выберите **Cookies**, и в открывшейся панели можно будет увидеть ваш сайт (рис. 13.15). Выберите этот сайт, в центре появятся все Cookie-значения, в нашем случае — это `.MyWebSite.Session`.

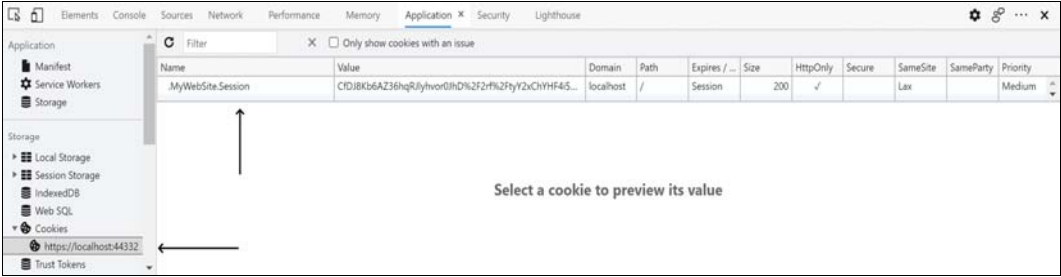


Рис. 13.15. Cookie-сессии

Зачем нужно Cookie-значение? Допустим, 100 человек обращаются к сайту — как сервер сможет различить каждого из них? Для сервера каждый запрос выглядит анонимным, и чтобы различить пользователей, каждому из них присваивается уникальное значение, которое сохраняется в Cookie, и сервер по этому значению может отличить одного пользователя от другого.

ПРИМЕЧАНИЕ
Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Session` сопровождающего книгу электронного архива (см. приложение).

13.14. Cookie

Сессии — отличный способ хранить информацию на короткий срок, но после перезапуска браузера придется снова восстанавливать данные, а в случае с авторизацией придется каждый раз заходить на сайт.

Cookie же можно устанавливать так, чтобы они сохранялись в системе на долгое время и даже после перезапуска компьютера.

Чтобы установить Cookie, нужно добавить его в список Cookies-ответа. Ответ этот находится в свойстве `Response`:

```
Response.Cookies.Append("usercookie", model.Email,
    new CookieOptions()
    {
        Path = "/",
        Expires = DateTimeOffset.Now.AddDays(10)
    }
);
```

Свойство `Response` — это класс `HttpResponse`, отвечающий за все, что мы хотим направить пользователю. Помимо текста страницы, мы можем отправлять еще и `Cookie`-значения, которые будут говорить браузеру, что их нужно запомнить. При добавлении `Cookie` мы должны указать три параметра: имя, значение и параметры в виде объекта класса `CookieOptions`. Все эти параметры очень важны, потому что если ничего не указать, то значение может работать неожиданным образом.

В параметрах обязательно следует указать путь, для которого действует значение. Например, указав `/`, мы сообщаем браузеру клиента, что он должен отправлять это значение нам всегда, когда пользователь обращается к сайту. Если указать определенную папку, то `Cookie`-значение будет отправляться только для этой папки, и это реально имеет смысл, потому что они будут отправляться с абсолютно каждым запросом. Чем больше вы значений добавляете, тем больше данных будет сопровождать каждый запрос от браузера к серверу.

Если какое-то значение нужно лишь в определенной области сайта, то имеет смысл ограничить `Cookie` только определенным URL.

Второй параметр — это время жизни `Cookie`. По умолчанию эти значения умирают после закрытия браузера, т. е. работают практически так же, как и сессии. Разница только в том, что значения в сессии хранятся на сервере, — просто они привязаны к определенному `Cookie` уникальному значению, которое идентифицирует пользователя. А вот значение в `Cookie` хранится в самом `Cookie` и, как мы только что отмечали, будет отправляться серверу с каждым запросом, но умрет после завершения сессии.

Чтобы `Cookie` не умирало при закрытии браузера, а продолжало жить, надо указать время его жизни, поэтому параметр `Expires` также очень важен.

Когда сервер сохранил данные, мы можем их прочитать из свойства запроса `Request`. В этом свойстве хранится информация о том, что пользователь направляет на сервер.

Давайте прочитаем имя пользователя из `Cookie`:

```
if (HttpContext.Request.Cookies.ContainsKey("usercookie"))
{
    username = HttpContext.Request.Cookies["usercookie"];
}
```

Учитывая, что мы сейчас работаем с тестовым примером, я сохраняю имя в «чистом виде». В реальном приложении я никогда бы не посоветовал вам так делать, потому что пользователь может просмотреть `Cookie`-значения в браузере и даже

изменить их. На рис. 13.15 показано Cookie-значение сессии, и точно так же будут храниться и любые другие Cookie, которые вы будете создавать.

Никогда не сохраняйте ничего важного в Cookie и не доверяйте этим значениям — их пользователь может изменить.

Чтобы удалить Cookie, можно изменить время `Expires` так, чтобы его значение было уже устаревшим, или просто удалить его из списка:

```
[Route("Logout")]
public IActionResult Logout()
{
    Response.Cookies.Delete("usercookie");
    return Redirect("/");
}
```

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Cookie` сопровождающего книгу электронного архива (см. *приложение*).

13.15. Доступ к запросу

До сих пор мы использовали переменные и объекты, данные в которые попадают автоматически, потому что .NET пытается связывать переменные с параметрами по имени.

С другой стороны, мы уже познакомились с *запросом* — свойством контроллера `Request`, через которое мы можем получить доступ ко всей возможной информации о запросе. Через это свойство мы обращались к Cookie-значениям, и здесь же присутствуют и все параметры, которые пользователь отправил на сервер.

Если форма направлена на сервер методом POST, то параметры будут записаны в `Request.Form`. Это список всех параметров — здесь найдутся и те, которые фреймворк смог назначить переменным, и те, которые не смог.

По возможности используйте автоматический процесс, но если есть какой-либо уникальный случай и автоматическое назначение значений использовать не получается, то к E-mail значению в отправленной на сервер форме можно обратиться так:

```
Request.Form["Email"].ToString()
```

Если запрос на сервер поступил методом GET, то параметры будут записаны в другое свойство — `Request.Query`.

Давайте сделаем так, чтобы при загрузке формы для входа на сайт, если в URL есть значение E-mail, использовалось по умолчанию именно оно:

```
[HttpGet]
public IActionResult Index()
{
    var model = new LoginModel();
```

```
model.Email = Request.Query.ContainsKey("Email") ?  
    Request.Query["Email"] : "";  
ViewBag.Title = "GET запрос";  
return View(model);  
}
```

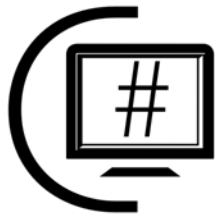
Здесь логика простая: если в свойстве `Request.Query` есть параметр с именем `Email`, то использовать его. Иначе — пустую строку.

Теперь, если загрузить страницу `https://localhost:7261/Login?email=test@email.com`, то на форме в поле **E-mail** уже будет находиться указанный в качестве параметра `Email`.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter13\Request` сопровождающего книгу электронного архива (см. *приложение*).

ГЛАВА 14



Базы данных

Работа с базами данных, наверное, самая популярная и самая востребованная в программировании область. Знание баз данных нужно практически в любой области: при разработке настольных приложений, при создании веб-сайтов и даже в играх бывает необходимость хранить какую-то информацию в базе данных.

В моей профессиональной деятельности программиста чаще всего приходится работать именно с базами данных: от небольших таблиц до гигабайтных хранилищ. Базы данных бывают разные, но это не значит, что нам нужно знать их все до единой и уметь пользоваться их особенностями. Существует несколько технологий, которые обеспечивают доступ к разным серверам баз данных, не зависящим от производителя сервера. В .NET основной библиотекой для работы с базами данных является ADO.NET.

В начале этой главы мы потратим немного времени на достаточно низкоуровневое программирование, и может показаться, что работа с базой данных слишком сложная, но на самом деле есть несколько библиотек, которые делают программирование баз данных проще. Но прежде чем использовать такие библиотеки, лучше разобраться, как все работает на низком уровне.

В этой главе мы будем создавать множество классов, и, возможно, вам покажется все очень сложным, но я хочу показать здесь некоторые важные и полезные приемы, которые пригодятся в реальной работе. Если для предыдущих изданий я пытался сделать код как можно проще, то в этом издании я во многом переписал материал главы заново и постарался на простых примерах показать, как делать еще и правильно.

14.1. Библиотека ADO.NET

Библиотека ADO.NET (Active Data Object .NET) — это набор классов, предназначенных для взаимодействия с различными хранилищами данных (базами данных). С помощью ее классов вы можете подключиться к серверу, сформировать и направить серверу запрос, получить результат и обработать его.

За подключение и непосредственную работу с базами данных отвечают поставщики данных. В .NET есть два поставщика данных: SQL Client .NET Data Provider и OLE DB .NET Data Provider. Первый из них предназначен для работы только с базами данных Microsoft SQL Server. За счет узкой направленности на одну базу данных от одного производителя классы и код провайдера могут быть оптимизированы для максимально эффективной работы с сервером.

Для прочих баз данных Microsoft не стала создавать отдельных провайдеров, как она сделала для Microsoft SQL Server. Производители баз данных могут сами написать свои библиотеки для оптимизированного доступа к данным. Вместо специализированных под каждую базу данных провайдеров, Microsoft реализовала универсальный провайдер OLE DB .NET Data Provider, позволяющий подключиться к любой базе данных, для которой есть поставщик данных OLE DB. В настоящее время такие поставщики есть для большинства баз данных.

Так как Microsoft SQL Server имеет OLE DB-драйвер, то к этой базе можно подключаться с помощью любого из двух упомянутых провайдеров. Конечно же, SQL Client .NET Data Provider работает лучше, но OLE DB .NET Data Provider позволяет создавать универсальный код, который сможет работать с любыми базами данных.

Если нет особой необходимости поддерживать разные базы данных и у вас в настоящее время принято решение работать с MS SQL Server, то я, разумеется, рекомендую использовать классы, специализированные для MS SQL Server.

Чтобы разделить классы, они находятся в разных пространствах имен. Классы для работы с SQL Client .NET Data Provider — в пространстве имен `System.Data.SqlClient` и `Microsoft.Data.SqlClient`, а классы OLE DB .NET Data Provider — в `System.Data.OleDb`.

Почему для доступа к MS SQL Server предусмотрены два пространства имен? Первое пространство — `System.Data.SqlClient` — это старые классы, которые уже больше не будут обновляться, и вместо них Microsoft рекомендует использовать `Microsoft.Data.SqlClient`. Недавно у нас на работе обнаружился один странный баг, который приводил к исключительной ситуации, и, что интересно, простая замена пространства имен решила проблему, потому что в новом коде эту ошибку уже исправили.

В целом классы из пространств имен `System.Data.SqlClient` и `Microsoft.Data.SqlClient` работают одинаково, одинаково реализованы и все имена классов — простая замена пакета пространств имен и перекомпиляция позволят перейти на новую библиотеку.

В библиотеке имена классов `OleDb` различаются, но для удобства разработки классы обоих провайдеров реализованы схожим образом и наследуются от одного и того же базового класса. Это значит, что и методы работы с данными идентичны. Для перевода кода с одного провайдера на другой достаточно изменить только имя класса. Например, за подключение к базе данных в провайдере SQL Client отвечает класс `SqlConnection`, а в провайдере OLE DB — класс `OleDbConnection`. Оба они являются потомками класса `DBConnection`, который реализует одинаковые функции и объявляет методы, независимые от провайдера. Заменяв имя класса `SqlConnection` на `OleDbConnection`, вы легко можете перейти с одного провайдера на другой.

Рассматривать оба интерфейса в одной книге — это лишняя трата времени, да и книга станет слишком толстой. Чтобы охватить максимально возможный материал и не потратить на это много бумаги, мы рассмотрим только современные классы MS SQL Server из пространства имен `Microsoft.Data.SqlClient`, потому что C#-программисты чаще всего в качестве базы данных используют MS SQL Server. Если вы будете использовать Oracle или другую базу, то простая замена фрагмента SQL в имени класса на `OleDb`, скорее всего, решит проблемы.

Для работы с примерами в этой главе вам понадобится MS SQL Server, который доступен для программистов бесплатно, — просто во время установки нужно указать, что вы устанавливаете версию для программиста.

14.2. Строка подключения

Для соединения с базой данных служит класс `OleDbConnection`. Этому классу нужно указать с помощью строки подключения (*Connection String*) параметры подключения к базе данных. Из этой строки `OleDbConnection` узнает, где находится база данных и какие параметры надо использовать для подключения и авторизации. Строку подключения можно написать самостоятельно вручную, а можно использовать встроенное в ADO окно создания строки.

Чтобы воспользоваться удобным окном создания строки подключения в Windows, создайте в любом месте на диске файл с расширением `udl`. Это можно сделать в Проводнике или в любом другом файловом менеджере. Имя файла и его расположение не имеют никакого значения, главное — это расширение. Попробуйте запустить созданный файл, и перед вами откроется диалоговое окно **Data Link Properties** (Свойства связи с данными) для редактирования строки подключения (рис. 14.1). Давайте рассмотрим его чуть поближе.

Для начала переключитесь на первую вкладку: **Provider** (Поставщик данных). Здесь поставщик данных — это не те поставщики, которых мы рассматривали в *разд. 14.1* (OLE DB и SQL Client), — на вкладке представлены драйверы, которые будут использоваться поставщиком данных .NET. Понимаю, запутанно, но попробую уточнить. Мы используем поставщиков данных OLE DB и SQL Client в .NET для того, чтобы подключаться к базе данных и работать с данными. А поставщики .NET не могут, не умеют и не хотят работать с базой данных напрямую. И в соответствии с известным армейским заветом: «не можешь — научим, не хочешь — заставим» нам придется их учить. А вот учить можно с помощью поставщиков данных базы данных. Этими поставщиками на самом деле являются драйверы, которые реально умеют работать с СУБД и станут посредниками между базой данных и поставщиком данных .NET.

Так как мы будем использовать родные для SQL Server классы, выбираем **SQL Server Native Client**, нажимаем кнопку **Next** и переходим на вторую вкладку окна свойств связи с данными — **Connection** (Подключение) — для создания строки подключения. Ее содержимое зависит от используемой базы данных. В случае с подключением к базе данных это окно будет выглядеть так, как показано на рис. 14.2.

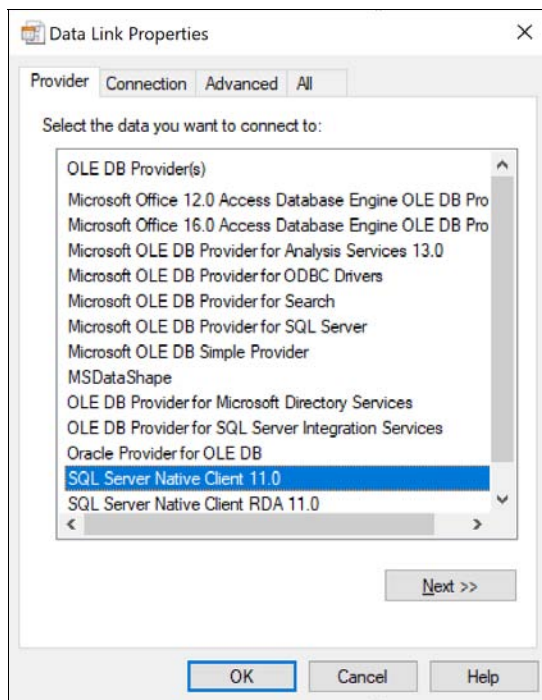


Рис. 14.1. Вкладка **Provider** окна **Data Link Properties** для выбора поставщика данных

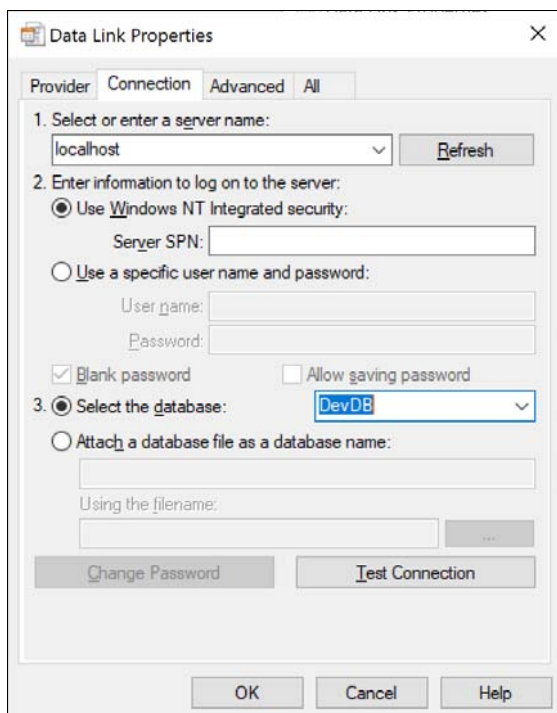


Рис. 14.2. Настройка подключения к базе данных

Здесь нужно выбирать имя сервера, пользователя, пароль и имя базы данных. Если вы используете сервер, установленный локально, то, скорее всего, в качестве имени сервера можно указать `localhost` и выбрать **Use Windows NT Integrated Security**.

Нажмите кнопку **ОК**, чтобы сохранить строку подключения. Но куда? В файл, который вы создали. Откройте ваш файл с расширением `udl` с помощью любого текстового редактора — например, Блокнота. В моем случае содержимое файла оказалось следующим:

```
[oledb]
; Everything after this line is an OLE DB initstring
Provider=SQLNCLI11.1;Integrated Security=SSPI;Persist Security
Info=False;User ID="";Initial Catalog=DevDB;Data Source=localhost;Initial
File Name="";Server SPN=""
```

Первая строка представляет собой начало раздела, указывающее на то, что перед нами строка подключения OLE DB. Вторая строка — комментарий. На то, что это комментарий, указывает точка с запятой в начале строки. А вот третья строка — это сама строка подключения, которую можно один в один копировать в ваш код и применить в программе.

Использование окна вида, показанного на рис. 14.2, очень удобно в случае подключения к базам данных OLE DB. В случае же с MS SQL Server строка подключения может быть намного проще, если создавать ее вручную. Например, если мы хотим подключиться под текущим пользователем, то строка подключения будет выглядеть так:

```
Data Source=СЕРВЕР;Initial Catalog=БАЗА;Integrated Security=True
```

А если вы хотите использовать определенного пользователя, то так:

```
Data Source=СЕРВЕР;Initial Catalog=БАЗА;Integrated Security=False;
User Id=ИМЯ_ПОЛЬЗОВАТЕЛЯ;Password=ПАРОЛЬ
```

Здесь:

- ☐ СЕРВЕР — адрес сервера, и при использовании локального компьютера это может быть `localhost`;
- ☐ БАЗА — имя базы данных на сервере;
- ☐ ИМЯ_ПОЛЬЗОВАТЕЛЯ — имя пользователя базы данных;
- ☐ ПАРОЛЬ — само имя говорит, что это пароль.

14.3. Подключение к базе данных

В качестве основы для примеров этого приложения я воспользуюсь кодом сайта, который мы писали в *главе 13*. Откройте проект — продолжим его улучшать.

Прежде чем использовать код для работы с базой данных, нужно установить соответствующий пакет. В окне Проводника решения щелкните правой кнопкой мыши на **Dependencies** (Зависимости) и выберите **Manage NuGet Packages** (Управление

пакетами NuGet) — в основной области среды разработки должна появиться вкладка **Browse** (рис. 14.3).

Перейдите на вкладку **Browse** и в строку поиска введите `SqlClient`. Нам нужен будет именно `Microsoft.Sql.Client`. Выделите этот пакет, и справа должна появиться панель с выпадающим списком, где можно выбрать версию и увидеть список доступных классов. Нажмите кнопку **Install**, чтобы подключить пакет.

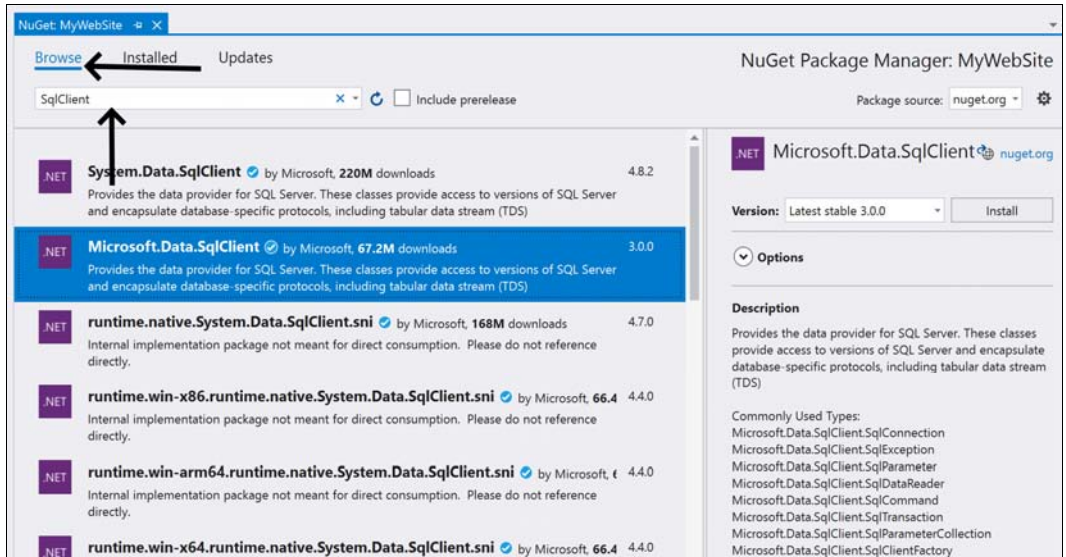


Рис. 14.3. Окно установки пакетов

Чтобы работать с провайдером OLE DB, нужно ввести `OleDb`, и в результате вы должны увидеть пакет `System.Data.OleDb`. Как я уже говорил, мы будем рассматривать только `Microsoft.Sql.Client`.

Что на самом деле произошло, когда мы подключили новый пакет? Откройте файл проекта или щелкните на имени проекта в Проводнике решения, чтобы увидеть исходный код:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net7.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="3.0.0" />
  </ItemGroup>
</Project>
```

У нас появилась новая строка `PackageReference`, где ее атрибут `Include` указывает на пакет, который мы подключили, а атрибут `Version` — на версию. Не используя окно управления пакетами, мы могли бы добавить эту строку в файл проекта вручную, и результат был бы тем же самым, но это происходит не всегда. Некоторые пакеты

могут выполнять дополнительные действия, а не только изменять файл проекта, да и с помощью окна управления пакетами удобнее искать нужный.

Если компилятор не может найти какой-то класс, то вы можете открыть окно управления пакетами, попробовать поискать по имени класса и найти отсутствующий пакет.

Итак, нужный пакет установлен, и нам теперь доступны классы для работы с базой данных. Чтобы обращаться к базе данных: получать данные или изменять их, нужно соединение, и за него отвечает класс `SqlConnection`. Самый простой способ создать это подключение — инициализировать объект такого класса и передать конструктору строку подключения:

```
SqlConnection connection = new SqlConnection(  
    "Server=localhost;Database=testdb;Trusted_Connection=True;");
```

Этот код вполне рабочий, но использовать строку подключения прямо в коде — плохо. Очень часто программисты, работая над кодом, держат базу данных локально на своем компьютере. Однако на сайтах в реальности база данных и код приложения, как правило, находятся на разных компьютерах, и нужно иметь возможность изменить строку подключения в любой момент, чтобы указать другое расположение сервера базы данных.

Если же мы расположим конфигурацию прямо в коде, то без перекомпиляции приложения изменить строку подключения не получится. Чтобы можно было менять что-либо без перекомпиляции, мы должны поместить это в конфигурационный файл. И у веб-приложений такой файл есть — `appsettings.json` (мы с ним работали в разд. 13.2). Откройте этот файл и добавьте в него следующий код:

```
"ConnectionStrings": {  
    "DefaultConnection":  
        "Server=localhost;Database=testdb;Trusted_Connection=True;"  
}
```

Строки подключения принято помещать в секцию `ConnectionStrings`. Вы можете просто создать параметр без секции, но тогда читать его придется как простой параметр. Если же мы поместим этот параметр в секцию `ConnectionStrings`, то сможем получить доступ к этой конфигурации через специализированный метод `GetConnectionString`, которому нужно передать имя параметра:

```
configuration.GetConnectionString("DefaultConnection");
```

Этот метод будет искать строку подключения с именем `DefaultConnection` в секции `ConnectionStrings`.

Отлично, теперь мы можем создавать соединение следующим способом:

```
SqlConnection connection = new SqlConnection(  
    configuration.GetConnectionString("DefaultConnection"));
```

14.4. Инъекция зависимостей на примере подключений

Мы все ближе к реальному программированию... Но сначала надо еще немного отойти от конфигурации. Не очень хорошо, если мы каждый раз при обращении к базе данных будем явно создавать соединение. Лучше было бы уйти от этого и иметь какой-то способ, где мы будем говорить: дайте мне соединение с базой данных, и нам должно быть все равно, как оно создается и как работает.

Использование `SqlConnection` — это достаточно низкоуровневый подход, и чтобы каждый раз не создавать соединение в своем коде, я предпочитаю абстрагироваться от него с помощью специального класса. Такой прием, как *инъекция кода*, достаточно широко используется в веб-программировании на C#, и вот настало время попрактиковаться в самостоятельном создании и использовании такой инъекции.

Для абстракции нам понадобится интерфейс `IDbConnection` и класс `DbConnection`, который реализует этот интерфейс. Файл для интерфейса создаем в папке `Model\Interfaces`, а файл для класса — в папке `Model\Implementation`. Структура моего проекта показана на рис. 14.4.

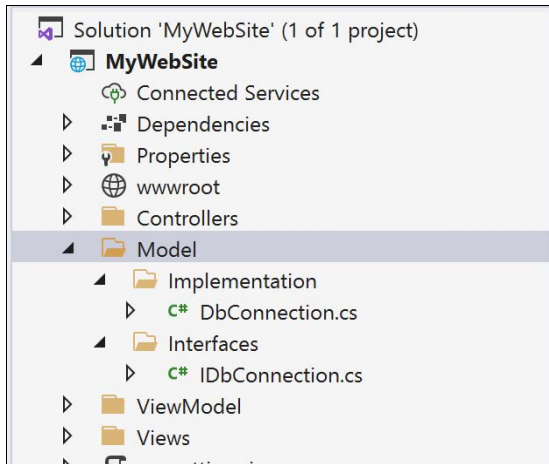


Рис. 14.4. Структура проекта

Код интерфейса простой — у него должен быть только один метод создания соединения:

```
using Microsoft.Data.SqlClient;

namespace MyWebSite.Model.Interfaces
{
    public interface IDbConnection
    {
        SqlConnection CreateConnection();
    }
}
```


Класс, который реализует этот интерфейс, может выглядеть так:

```
using Microsoft.Extensions.Configuration;
using Microsoft.Data.SqlClient;
using MyWebSite.Model.Interfaces;

namespace MyWebSite.Model.Implementation
{
    public class DbConnection : IDbConnection
    {
        IConfiguration configuration;
        public DbConnection(IConfiguration configuration) {
            this.configuration = configuration;
        }

        public SqlConnection CreateConnection() {
            SqlConnection connection new SqlConnection(
                configuration.GetConnectionString("DefaultConnection")
            );
            connection.Open();
            return connection;
        }
    }
}
```

Класс в качестве конструктора получает что-то, что реализует интерфейс `IConfiguration`. Метод `CreateConnection` создает соединение, используя строку подключения из конфигурации.

Когда мы смотрели на пример с конфигурационными файлами, то я говорил, что для работы с конфигурацией нужен интерфейс `IConfiguration`. Стоит только указать его в качестве параметра класса, и за счет инъекции кода фреймворк сам подставит класс, который реализует этот интерфейс.

Отлично, но как теперь создавать наш класс `DbConnection`? Конструктор класса получает `Iconfiguration`, а как нам инициализировать его, чтобы передавать конфигурацию:

```
IDbConnection connection = new DbConnection(...);
```

Для этого нам нужно иметь переменную типа `IConfiguration`, которую и передавать в этот конструктор.

Так как инъекция в основном происходит через конструкторы, что если мы создадим свой контроллер и у его конструктора скажем, что нам нужен экземпляр класса `IDbConnection`? В следующем примере я создал новый контроллер `DbTestController` с конструктором, который принимает в качестве параметра `IDbConnection`. Как вы думаете, .NET сможет догадаться, что при создании контроллера нам нужно автоматически предоставить экземпляр класса, который реализует интерфейс `IDbConnection`, как показано в следующем коде?

```
using Microsoft.AspNetCore.Mvc;
using MyWebSite.Model.Interfaces;

namespace MyWebSite.Controllers
{
    public class DbTestController : Controller
    {
        IDbConnection db;
        public DbTestController(IDbConnection db)
        {
            this.db = db;
        }

        public string Index()
        {
            using (var connection = db.CreateConnection())
            {
                return connection.State.ToString();
            }
        }
    }
}
```

Пробуем обратиться к этому контроллеру через `\dbtest\index`, но, к сожалению, автоматическая инъекция не срабатывает, и этот код завершается ошибкой:

```
InvalidOperationException: Unable to resolve service for type
'MyWebSite.Model.Interfaces.IDbConnection' while attempting to activate
'MyWebSite.Controllers.DbTestController'.
```

Общий смысл этого сообщения об ошибке таков: Ошибка неверной операции, не могу определить сервис для типа `MyWebSite.Model.Interfaces.IDbConnection` во время попытки активации `MyWebSite.Controllers.DbTestController`.

Другими словами, когда активировался наш контроллер, фреймворк не смог определить, какой тип данных создать для интерфейса `IDbConnection`. Множество классов могут реализовывать этот интерфейс, и то, что по счастливой случайности у нас только один класс `DbConnection`, не дает фреймворку права что-то делать автоматически, — мы должны явно где-то сказать, что при попытке инициализации интерфейса `IDbConnection` требуется создавать экземпляры класса `DbConnection`.

Для встроенных типов, таких как конфигурация, это уже сделано за нас — Microsoft позаботилась о классах, которые сама написала. А о своих классах мы должны позаботиться самостоятельно.

Инъекция зависимостей в основном настраивается в файле `Program.cs`:

```
services.AddSingleton<Model.Interfaces.IDbConnection,
    Model.Implementation.DbConnection>();
```

С помощью `services.AddSingleton` я привязываю интерфейс `IDbConnection` к классу `DbConnection`. Вот теперь фреймворк знает, что если какой-то класс в конструкторе

просит интерфейс `IDbConnection`, значит, для него нужно автоматически создать `DbConnection`.

Так что если сейчас запустить этот код и обратиться к `ldbtest\index`, то мы должны увидеть на странице надпись **Open**.

Но вернемся к классу контроллера. В методе `Index` у нас достаточно простой код:

```
using (var connection = db.CreateConnection())
{
    return connection.State.ToString();
}
```

Любой доступ к базе данных лучше писать только из модели (бизнес-логики), но мы сейчас просто учимся и писать еще одну абстракцию пока не станем, а просто протестируем класс, который отвечает за подключение.

Тут стоит отметить два очень важных момента:

- ❑ мы теперь не должны думать о конфигурации, не должны думать, как создается строка соединения, — мы просто вызываем метод `CreateConnection()` и получаем готовый объект для работы с базой данных;
- ❑ мы используем ключевое слово `using` — чтобы убедиться, что соединение закроется по завершении работы с этим кодом.

Для связывания интерфейсов с классами есть три метода:

- ❑ `AddSingleton` — при его использовании реализуется подход `Singleton`, когда в приложении создается один-единственный объект, и каждый раз, когда мы запрашиваем конфигурацию, будет возвращаться один и тот же объект.

Этот подход можно задействовать в тех случаях, когда у класса нет свойств, которые сохраняют какое-то состояние. Следующие признаки свидетельствуют, что класс может использоваться как `Singleton`: у класса есть только методы, или класс содержит глобальную и неменяемую информацию. В нашем случае конфигурация никак не меняется и работает на протяжении всей работы приложения;

- ❑ `AddScoped` — при его использовании для каждого нового запроса на сервере будет создаваться новый объект класса. Этот вариант подходит в том случае, когда в объекте могут сохраняться данные пользователя. Например, если в объекте сохраняется имя текущего пользователя в качестве одного из свойств, то для нового запроса нельзя снова использовать тот же самый объект, потому что это может быть другой пользователь;
- ❑ `AddTransient` — при каждом обращении к интерфейсу будет создаваться новая версия объекта. Объекты могут создаваться часто и много. Но они каждый раз будут новые, и каждый раз можно в них хранить разные данные. Из личного опыта знаю, что мне такое бывает нужно очень редко.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке `Source\Chapter14\Connection` сопровождающего книгу электронного архива (см. приложение).

14.5. Пул соединений

Открытие и закрытие соединений с базой данных — очень дорогое удовольствие с точки зрения производительности. В момент инициализации подключения клиенту требуется выполнить весьма много действий, скрытых от конечного пользователя. Может быть, открыть соединение только один раз и потом держать его постоянно открытым на протяжении всего времени выполнения программы? Это решение можно рассмотреть, но оно не всегда является идеальным.

Держать постоянно открытое соединение может быть накладно и для сервера. Простые серверы баз данных очень часто ограничены количеством одновременно поддерживаемых подключений. Нагрузка на серверы может влиять на определение нужного количества таких соединений в определенный момент: сейчас на сайте может быть пять пользователей, и достаточно даже одного подключения, которое будет использоваться по очереди, а через час на сайт может зайти тысяча посетителей, и нужно будет уже как минимум 100 подключений.

В случае с десктопными приложениями держать активное соединение совсем уж невыгодно. Например, пользователь выбрал данные для редактирования и открыл соответствующее окно. Он может держать это окно открытым полчаса или уйти на обед с запущенной программой, так зачем же держать соединение активным? Если у вас нет постоянного обмена данными с сервером, то все время держать активное подключение не имеет смысла.

Тут нужно заметить, что на активное подключение нет тайм-аута, по которому оно могло бы разрываться. Есть тайм-аут на процесс установки соединения, а если соединение уже установилось, то оно может быть активным неделями. Поэтому некоторые пользователи не закрывают программы. Да я и сам не выключаю программы и компьютер на работе, а просто гашу монитор и ухожу.

Если вы решили закрывать соединение сразу после обработки данных, то не стоит бояться, что произойдет сильное падение производительности при частом открытии/закрытии соединения с сервером баз данных. Основные потери происходят по двум статьям:

- ❑ *поиск сервера по имени.* Прежде чем соединиться с компьютером, программа должна найти его адрес. В Интернете по имени сервера ищется IP-адрес с помощью протокола DNS, а в локальных сетях соединение происходит по MAC-адресу, который ищется с помощью протокола ARP (Address Resolution Protocol, протокол определения адреса). После первой попытки соединиться информация об адресе сохраняется в кеше, поэтому последующие вызовы не тратят драгоценное время на повторное определение адреса;
- ❑ *непосредственная установка соединения и выделение ресурсов, необходимых для поддержки этого соединения.* Эта проблема легко решается с помощью пула соединений. Я даже скажу больше — она уже решена, и вам не нужно писать ни строчки кода. Дело в том, что каждый поставщик данных ADO.NET уже реализует пул.

Когда вы уничтожаете объекты класса `Connection`, то поставщик данных реально не закрывает соединение с базой данных. Объект помечается как неиспользуемый, и если в течение определенного времени клиент снова запросит подключение, то будет использоваться уже существующее соединение, которое было помечено как неиспользуемое. Таким образом, потери на открытие объекта `Connection` будут минимальными даже при очень частом открытии соединений с сервером баз данных с помощью:

```
connection.Open()
```

Если вы не хотите, чтобы подключение попадало в пул соединений драйвера, а решите сами сделать что-то подобное или вообще откажетесь от услуг кеширования соединения, то об этом нужно сообщить драйверу через строку подключения. Для провайдера OLE DB в строку подключения тогда нужно добавить параметр:

```
OLE DB Services=-4
```

А для подключения к SQL Server (класс `SqlConnection`) в строке подключения нужно указать параметр:

```
Pooling=false
```

14.6. Чтение данных из БД

Для выполнения команд используются объекты класса `OleDbCommand`. У конструктора нет параметров — достаточно просто проинициализировать объект значением по умолчанию. После этого в свойство `CommandText` надо поместить SQL-запрос, и можно его выполнять.

Для выполнения запросов существует несколько методов. Все зависит от того, какой результат вы хотите получить. Давайте начнем с чтения таблицы.

С помощью SQL Management Studio или любой другой программы для работы с MS SQL Server откройте из папки `Source\Chapter14\` сопровождающего книгу электронного архива файл `DBBackup.sql`, который создает базу данных `testdb` и необходимую для этой книги структуру таблиц. Одна из таких таблиц там — это таблица `City`, состоящая из двух колонок: `CityID` и `CityName`.

Давайте создадим бизнес-модель, с помощью которой мы сможем работать с этой таблицей. Для начала создадим класс `City` из двух полей: `CityID` и `CityName` — как у базы данных:

```
namespace MyWebSite.Model.DataModel
{
    public class City
    {
        public int? CityId { get; set; }
        public string CityName { get; set; }
    }
}
```

Я поместил этот класс в отдельную папку и пространство имен `DataModel`.

Теперь создаем интерфейс `ICityProvider`, который описывает необходимый нам провайдер для доступа к данным. Пока в нем будет только один метод — `GetCities()`:

```
using System.Collections.Generic;
using MyWebSite.Model.DataModel;

namespace MyWebSite.Model.Interfaces
{
    public interface ICityProvider
    {
        Task<List<City>> GetCities();
    }
}
```

Ну и, наконец, сама реализация интерфейса будет выглядеть так:

```
public class CityProvider: ICityProvider
{
    IDbConnection connection;
    public CityProvider(IDbConnection connection)
    {
        this.connection = connection;
    }

    public async Task<List<City>> GetCities()
    {
        List<City> result = new List<City>();
        using (var connection = this.connection.CreateConnection())
        {
            SqlCommand command = connection.CreateCommand();
            command.CommandText = "select CityId, CityName from city";
            using (var reader = await command.ExecuteReaderAsync())
            {
                while (await reader.ReadAsync())
                {
                    result.Add(new City()
                    {
                        CityId = reader.GetInt32(0),
                        CityName = reader.GetString(1)
                    });
                }
            }
        }
        return result;
    }
}
```

Рассмотрим этот код подробнее.

Провайдер `CityProvider` будет получать в качестве параметра конструктор класса для соединения с базой данных. Его мы получим автоматически через инъекцию кода.

Единственный метод `GetCities()` возвращает список городов. Здесь мы используем объект подключения к базе данных, чтобы открыть соединение с ней, и тут начинается самое интересное:

```
SqlCommand command = connection.CreateCommand();  
command.CommandText = "select CityId, CityName from city";
```

У объекта подключения `SqlConnection` есть метод `CreateCommand`, возвращающий объект типа `SqlCommand`, через который мы можем обращаться к базе данных. Команду, которую мы хотим выполнить в базе данных, нужно поместить в свойство `CommandText` — в моем случае это простой SQL-запрос выборки всех данных из таблицы `City`.

Указав команду, мы можем теперь ее выполнить, а у `SqlCommand` есть несколько методов для выполнения команд — в зависимости от того, какой результат мы хотим получить. В нашем случае это могут быть несколько записей, а для такого случая существует метод `ExecuteReader` или его асинхронная версия — `ExecuteReaderAsync`. Я в этом примере использую именно асинхронную версию, поэтому перед вызовом метода стоит `await`, а сам метод возвращает `async Task<>`. Это позволит процессору выполнять другие задачи, пока база данных не вернет результат нашего запроса.

В качестве результата методы `ExecuteReaderAsync` и `ExecuteReader` возвращают объект типа `SqlDataReader`, через который мы можем уже этот результат читать. Чтение очередной строки осуществляет метод `ReadAsync`. Поэтому здесь мы вызываем этот метод в цикле, где методы, которые начинаются с `Get`, в качестве параметра получают число — номер колонки из результата, которую мы хотим прочитать:

```
while (await reader.ReadAsync())  
{  
    // читаем поля через reader.GetXxxxx()  
}
```

Мы читаем таблицу из двух колонок: `CityId` (число) и `CityName` (строка), и в зависимости от типа колонки должны выбирать соответствующий метод для чтения данных. Для чтения числа из цифровой колонки (`CityId`) я использую метод `GetInt32`, а для чтения строки — метод `GetString`. Разумеется, существуют методы для чтения всех типов данных, которые могут использоваться в базе данных.

Читая данные, я создаю новый экземпляр класса — `City` — и копирую в него данные.

Итак, если пробежаться по всем классам, которые задействованы в процессе:

- ❑ `SqlConnection` — отвечает за создание соединения с базой данных и создает команды `SqlCommand`, с помощью которых мы можем направлять на сервер SQL-запросы;
- ❑ `SqlCommand` — отвечает за запросы, которые будут направляться серверу;
- ❑ `SqlDataReader` — позволяет прочитать результат выполнения команды.

Если отказаться от асинхронности, то метод выглядел бы следующим образом:

```
public List<City> GetCities()
{
    List<City> result = new List<City>();
    using (var connection = this.connection.CreateConnection())
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "select * from city";
        using (var reader = command.ExecuteReader())
        {
            while (reader.Read())
            {
                result.Add(new City()
                {
                    CityId = reader.GetInt32(0),
                    CityName = reader.GetString(1)
                }); ;
            }
        }
    }
    return result;
}
```

Как видите изменений не так много: изменился тип возвращаемых данных — теперь это просто список, а не `Task`. В выделенных строках исчезло слово `await`, и у вызывающего метода исчезло в конце слово `Async`. Я все же рекомендую вам ориентироваться на асинхронную версию, потому что она лучше использует возможности компьютера.

И последний штрих перед тем, как перейти к контроллеру, — нужно подсказать фреймворку, как делать инъекцию для нашего класса:

```
services.AddScoped<Model.Interfaces.ICityProvider,
    Model.Implementation.CityProvider>();
```

Теперь создадим новый контроллер `CityController.cs` со следующим кодом:

```
public class CityController: Controller
{
    ICityProvider cityProvider;
    public CityController(ICityProvider cityProvider)
    {
        this.cityProvider = cityProvider;
    }

    public async Task<IActionResult> ListCities()
    {
        var result = await cityProvider.GetCities();
        return View(result);
    }
}
```


Контроллер получает в конструкторе экземпляр `ICityProvider` провайдера. Как видите, в этом коде вообще ничего нет, что касалось бы базы данных. Есть только интерфейс, который возвращает города, и контроллер совершенно ничего не знает о том, как провайдер получает и возвращает данные об этих городах.

Раз метод возвращает задачу, то мы должны обязательно использовать `await`, и контроллер тоже должен возвращать задачу, чтобы фреймворк мог корректно дождаться завершения работы.

Получив от провайдера города, я их передаю представлению, код которого выглядит так:

```
@model List<MyWebSite.Model.DataModel.City>

<div class="list">
    @foreach (var city in Model) {
        <div class="listitem">@city.CityName</div>
    }
</div>
```

14.7. Запросы с параметрами

В предыдущем разделе мы научились читать данные и выводить список всех городов. Допустим, мы теперь хотим создать форму для редактирования данных.

Давайте обновим представление и создадим две ссылки: для редактирования города и для его удаления:

```
<div class="list">
    @foreach (var city in Model) {
        <div class="listitem">
            <span>@city.CityName</span>
            <span><a href="/City/Edit/@city.CityId">Редактировать</a></span>
            <span><a href="/City/Delete/@city.CityId">Удалить</a></span>
        </div>
    }
</div>
```

Начнем с редактирования — для него придется написать чуть больше кода, но будет интересно. Для редактирования мы станем вызывать URL `/City/Edit/@city.CityId`, а при использовании маршрутизации по умолчанию это значит, что нам надо создать метод `Edit` в контроллере `CityController`:

```
[HttpGet]
public async Task<IActionResult> Edit(int id)
{
    var result = await cityProvider.GetCity(id);
    return View(result);
}
```

До сих пор мы писали код, начиная с бизнес-логики, но на этот раз начнем с представления. Новый метод `Edit` задействует провайдер, который мы уже использовали, но на этот раз вызывает метод `GetCity` и передает методу идентификатор, который мы получили в качестве параметра. Если вы не знаете, откуда появился параметр `id`, вернитесь к *разд. 13.5*, где мы это рассматривали.

Сейчас этот код компилироваться не станет, потому что у провайдера нет такого метода, и его нужно добавить. Сначала добавляем объявление метода в интерфейсе:

```
public interface ICityProvider
{
    Task<List<City>> GetCities();
    Task<City> GetCity(int id);
}
```

Теперь посмотрим на реализацию метода у класса `CityProvider`:

```
public async Task<City> GetCity(int id)
{
    using (var connection = this.connection.CreateConnection())
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText = "select * from city where cityid = @id";
        SqlParameter parameter = new SqlParameter()
        {
            ParameterName = "@id",
            Value = id,
            SqlDbType = System.Data.SqlDbType.Int
        };
        command.Parameters.Add(parameter);
        using (SqlDataReader reader = await command.ExecuteReaderAsync())
        {
            if (reader.HasRows && await reader.ReadAsync())
            {
                return new City()
                {
                    CityId = reader.GetInt32(0),
                    CityName = reader.GetString(1)
                };
            }
        }
    }
    return new City();
}
```

Точно так же, как и при отображении всех городов, я создаю здесь соединение и новую команду. Первое различие появляется в SQL-запросе:

```
command.CommandText = "select * from city where cityid = @id";
```

Этот SQL-запрос выбирает все колонки из таблицы `city`, где колонка `cityid` равна параметру `@id`. Имена параметров — это как переменные внутри SQL-запросов, и их имена начинаются с символа `@`. Имея имя параметра, мы должны предоставить теперь значение.

Для описания параметра в C# есть класс `SqlParameter`, которому нужно указать как минимум два свойства:

- ❑ `ParameterName` — имя параметра, которое мы дали в запросе SQL, в нашем случае это `@id`;
- ❑ `Value` — значение.

Желательно еще указать и тип `SqlDbType` — в нашем случае это `System.Data.SqlDbType.Int`. Тип должен совпадать с тем, что реально используется в базе данных. Типы баз данных отличаются от тех, что мы используем в C#, и их можно найти в перечислении `SqlDbType` пространства имен `System.Data`. Если вы укажете тип данных не точно, то не исключено, что код еще будет работать, но это может повлиять на производительность выполнения запроса на сервере:

```
SqlParameter parameter = new SqlParameter()  
{  
    ParameterName = "@id",  
    Value = id,  
    SqlDbType = System.Data.SqlDbType.Int  
};
```

Имея переменную `parameter`, мы должны добавить ее в список параметров свойства `Parameters` SQL-команды:

```
command.Parameters.Add(parameter);
```

Мы знаем, что результатом должна быть только одна строка, если она будет найдена, или ни одной строки, если пользователь передаст неверный идентификатор. Но с точки зрения кода это все еще таблица из двух колонок и одной строки, поэтому мы вызываем уже знакомый `ExecuteReaderAsync`.

Далее небольшое различие: прежде чем читать данные, я проверяю, ожидаются ли хоть какие-то строки в результате выполнения запроса. Это можно сделать, если посмотреть на свойство `HasRows` класса `SqlDataReader`.

Остальное вам уже должно быть знакомо: мы читаем данные из результата и возвращаем экземпляр класса `City` с данными о найденном городе.

Осталось только создать представление для нового действия `/city/edit`. Я сделал максимально простую HTML-форму, которая отображает найденное название города в текстовом поле, а в скрытом поле с именем `cityid` будет прятаться ID редактируемого города:

```
@model MyWebSite.Model.DataModel.City  
  
<div class="list">  
    <form method="post">  
        <input type="hidden" name="cityid" value="@Model.CityId" />
```

```

        <label>Город</label>
        <input name="CityName" value="@Model.CityName" />
        <button>Сохранить</button>
    </form>
</div>:

```

Эта форма умеет методом POST отправлять данные обратно на сервер. А в следующем разделе мы рассмотрим пример, как можно данные сохранить.

14.8. Редактирование данных

Здесь я приведу только код из провайдера, который непосредственно работает с базой данных. Остальной код попробуйте дописать сами, а если не получится, то можете использовать пример из папки `Source\Chapter14\Connection\` сопровождающей книгу электронного архива (см. *приложение*).

Для обновления данных мы можем использовать возможности SQL:

```

public async Task<bool> UpdateCity(City city)
{
    using (var connection = this.connection.CreateConnection())
    {
        SqlCommand command = connection.CreateCommand();
        command.CommandText =
            "update city set CityName = @cityname where cityid = @id";
        command.Parameters.AddWithValue("@id", city.CityId);
        command.Parameters.AddWithValue("@cityname", city.CityName);
        int rows = await command.ExecuteNonQueryAsync();
        return rows == 1;
    }
}

```

Как и в предыдущих случаях, я создаю соединение с базой данных, команду и добавляю SQL-запрос обновления данных. В тексте запроса вы можете увидеть два параметра: `@cityname` и `@id`.

В этот раз я решил параметры создать с помощью метода `AddWithValue`, которому нужно передать два параметра — имя SQL-параметра и его значение:

```
command.Parameters.AddWithValue("@id", city.CityId);
```

Тип указывать не нужно — его определяют по типу переменной, которую мы передадим в качестве второго параметра. Свойство `CityId` имеет числовой тип, а значит, в SQL будет передаваться параметр в виде числа.

Теперь главный вопрос: как выполнить этот запрос? Он не возвращает никаких данных, поэтому `ExecuteReaderAsync` использовать нельзя. Вместо него есть другие варианты: `ExecuteNonQuery` и `ExecuteNonQueryAsync`. Первый вариант — это выполнение SQL, который не является запросом на получение данных, синхронно, а второй делает то же самое, но асинхронно.

14.9. Транзакции

Транзакции отличаются тем, что все изменения в базе данных, сделанные внутри одной транзакции, будут выполнены полностью или не выполнены совсем. Каждое изменение в базе данных выполняется внутри какой-то транзакции, но если нужно сгруппировать несколько обновлений, то тут следует явно начинать и завершать транзакции. Если во время выполнения запросов внутри транзакции произойдет ошибка, то все изменения транзакции будут отменены.

За транзакцию отвечает класс `SqlTransaction`. Создать экземпляр этого класса можно с помощью метода `BeginTransaction`.

У этого класса есть следующие методы, которые могут нам пригодиться:

- ❑ `Commit()` — сохранить изменения, сделанные внутри транзакции;
- ❑ `Rollback()` — отменить изменения, т. е. откатить транзакцию.

Давайте рассмотрим использование транзакций на реальном примере. В листинге 14.1 программа в транзакции пытается выполнить запрос добавления данных в таблицу, но изменения не сохраняются, потому что в самом конце происходит откат транзакции.

Листинг 14.1. Использование транзакции

```
SqlTransaction transation = connection.BeginTransaction();
SqlCommand command = connection.CreateCommand();
command.Transaction = transation;
command.CommandText =
    "update city set CityName = @cityname where cityid = @id";
command.Parameters.AddWithValue("@id", city.CityId);
command.Parameters.AddWithValue("@cityname", city.CityName);
int rows = await command.ExecuteNonQueryAsync();
transation.Commit();
```

Смысл метода — создать объект класса `SqlTransaction`. После этого мы создаем команду и назначаем команде транзакцию, установив ее в свойство `Transaction`.

Теперь у нас есть возможность запомнить данные или отменить изменения. Для этого примера изменения запоминаются с помощью `Commit`.

Для работы с транзакциями есть еще один более простой подход — использовать класс `TransactionScope`. Этот класс реализует интерфейс `IDisposable`, и значит, его удобно использовать совместно с `using`:

```
using (TransactionScope scope = new TransactionScope())
{
    // код доступа к базе данных
    scope.Complete();
}
```

Здесь создается новый экземпляр класса `TransactionScope` со значениями по умолчанию, и транзакция объединит все, что находится в фигурных скобках. Чтобы зафиксировать изменения, нужно выполнить метод `Complete`. Если его не выполнить, то изменения не будут зафиксированы в базе данных.

Если внутри транзакции используется асинхронное программирование, то нужно использовать версию, у которой есть параметр, позволяющий указать поддержку асинхронности с помощью `TransactionScopeAsyncFlow`:

```
new TransactionScope(  
    TransactionScopeAsyncFlowOption.Enabled  
);
```

Здесь в качестве параметра я передал `TransactionScopeAsyncFlowOption.Enabled`, что позволяет включить поддержку асинхронного потока.

14.10. Библиотека **Dapper**

Взаимодействуя с базой данных через `SqlConnection` и `SqlCommand`, мы работаем на достаточно низком уровне, поэтому приходится выполнять много действий вручную, но такой подход позволяет достичь высокой производительности разработанных на его основе приложений.

Работая над сайтами с высокой нагрузкой, я нашел для себя идеальное сочетание скорости и удобства — библиотеку **Dapper**, которую написали создатели сайта [Stackoverflow](https://stackoverflow.com). Эта библиотека с открытым исходным кодом, и вы можете увидеть ее исходные коды на сайте <https://github.com/DapperLib/Dapper>.

Для работы с библиотекой нужно сначала подключить ее код через **Nuget package** или вручную. В первом случае в окне Проводника решения щелкаем правой кнопкой мыши на **Dependencies** и выбираем **Manage NuGet Packages**. На вкладке **Browse** ищем позицию **Dapper**, выделяем ее и в правой панели нажимаем кнопку **Install**.

В результате этих манипуляций в файле проекта `*.csproj` появилась новая строка:

```
<PackageReference Include="Dapper" Version="2.0.90" />
```

Вы могли бы записать туда эту строку вручную, и это был бы второй способ подключения библиотеки **Dapper**.

Помните, как много действий нам нужно было сделать, чтобы прочитать из базы данных все города. Мы должны были создать команду, потом читать данные из результата, для каждой строки создавать объект класса `City` и сохранять все это в массив.

Посмотрим, как это все делать с библиотекой **Dapper**. Открываем `CityProvider` и добавляем в самом начале пространство имен:

```
using Dapper;
```

Теперь у объекта `SqlConnection` появятся новые методы, имена которых начинаются с `Query`. Методы не появятся, пока вы не подключите пространство имен, так что этот шаг обязателен.

Теперь посмотрим, как будет выглядеть метод `GetCities` с использованием `Dapper`:

```
public async Task<List<City>> GetCities()
{
    using (var connection = this.connection.CreateConnection())
    {
        var result = await connection.QueryAsync<City>(
            "select CityId, CityName from City");
        return result.ToList();
    }
}
```

Здесь мы вызываем метод `QueryAsync`, которому передается SQL-запрос, а в угловых скобках записывается класс, список которого хотим получить. Всё — больше ничего делать не надо. После вызова `QueryAsync` мы получаем результат в виде `IEnumerable<City>`. Так как наш метод должен возвращать список, то я вынужден выполнить еще одну команду для превращения этого результата в список. Впрочем, можно было бы возвращать `IEnumerable`, и в реальном приложении я бы так и сделал.

Теперь посмотрим, как мы можем использовать запросы с параметрами и как бы выглядел при этом метод `GetCity`:

```
public async Task<City> GetCity(int id)
{
    using (var connection = this.connection.CreateConnection())
    {
        var result = await connection.QueryAsync<City>(
            "select CityId, CityName from City where CityId = @idparam",
            new { idparam = id });
        return result.FirstOrDefault();
    }
}
```

Параметры передаются в виде анонимного класса, где имя свойства — это имя параметра в запросе. Я переименовал имя параметра в запросе на `idparam`, чтобы оно отличалось от параметра, который мы получили в методе. И здесь мы снова привлекаем метод `QueryAsync`, который возвращает перечисление `IEnumerable`, но в этот раз с параметрами. Поскольку это перечисление, а нам нужен только один элемент, то я использую `FirstOrDefault` для получения первого элемента.

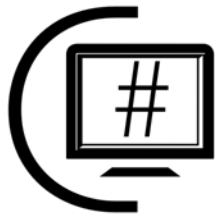
А в следующем примере мы используем другой метод из состава `Dapper` — `QueryFirstAsync`. Он сразу возвращает первый элемент из результата, и нам не нужно выполнять дополнительные действия:

```
public async Task<City> GetCity(int id)
{
    using (var connection = this.connection.CreateConnection())
    {
        return await connection.QueryFirstAsync<City>(
            "select CityId, CityName from City where CityId = @idparam ",
            new { idparam = id });
    }
}
```

Библиотека Dapper невероятно проста в использовании, делает код намного проще, и при этом скорость ее работы близка к прямому вызову API базы данных.

ПРИМЕЧАНИЕ

Исходный код примера к этому разделу можно найти в папке *Source\Chapter14\Dapper* сопровождающего книгу электронного архива (см. *приложение*).



Entity Framework Core

Когда Microsoft впервые выпустила Entity Framework Core, я не очень сильно впечатлился. Честно говоря, я до сих пор не сильно люблю использовать его для работы с базой данных в больших проектах. Просто я фанат чистого SQL, который позволяет мне получить всю гибкость, которую мы хотим.

Но не в моих правилах делить все на черное и белое, и, конечно же, я использую этот фреймворк. Получаю я от этого удовольствие? Нет! Особенно я не люблю его SQL-подобный синтаксис — я пробовал его изучать, но он мне не понравился. Так что я его не знаю и пока не хочу изучать. LINQ-подход я тоже не люблю, но, выбирая из двух зол, я выбрал для себя Entity Framework Core и буду здесь его использовать.

Далее я буду часто писать «Entity Framework» без явного указания «Core» или использовать сокращение EF, но все это будет в реальности иметь отношение именно к Entity Framework Core.

У Entity Framework нет особой магии, которая бы сделала доступ к базе быстрее, потому что это надстройка, которая под капотом все еще использует SQL-запросы для работы с реляционной базой данных. Она сама генерирует запросы в соответствии с тем, что мы пишем на C#, поэтому мы теряем гибкость и вынуждены работать с тем, что сгенерируют за нас.

Наверное, пришло время сказать, что же такое Entity Framework Core и для чего он нужен. Согласно Википедии это Object-Relational Mapping (ORM), а такие системы позволяют копировать данные из базы данных в объекты на C#, что очень удобно с точки зрения работы с данными. И если Dapper занимается только копированием данных в C#-объекты, то Entity Framework обладает намного большим количеством возможностей.

Entity Framework может работать с различными базами данных, хотя я буду все рассматривать с точки зрения MS SQL Server. С остальными базами работа будет организовываться так же — просто потребуется установить другой провайдер.

15.1. Создаем EF-приложение

В этой главе я буду показывать все на примере консольного приложения, чтобы не делать никаких визуальных интерфейсов, а сконцентрироваться только на коде.

Создав консольное приложение, нужно установить пакеты, с которыми мы будем работать. В принципе, их можно устанавливать по мере надобности, но я уже знаю, что нам пригодится, поэтому, прежде чем писать код, посмотрим на доступные пакеты и установим необходимые.

Вызываем пакетный менеджер: нажимаем правой кнопкой на имя папки **Dependencies** (Зависимости), выбираем **Manage NuGet Packages** и в поле поиска на вкладке **Browse** вписываем `EntityFrameworkCore` — чтобы увидеть все пакеты, которые относятся к EF Core (рис 15.1).

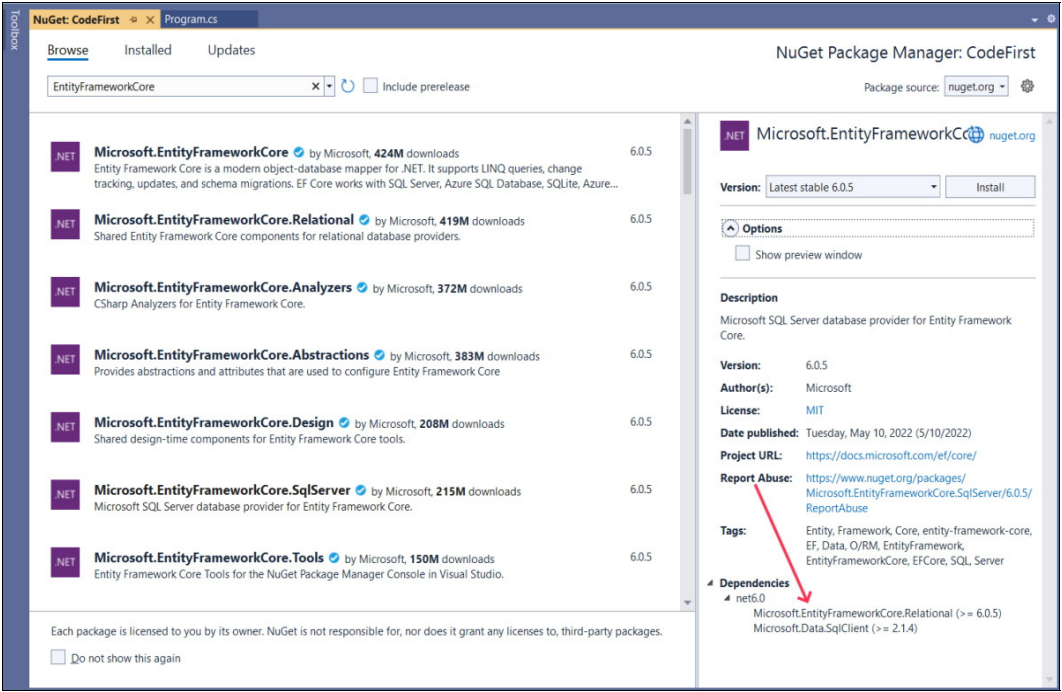


Рис. 15.1. Добавление пакетов EF

Можно начать с основы и установить **Microsoft.EntityFrameworkCore**, но в нем реализованы только базовые общие возможности. Установив его, мы не сможем подключиться к базе данных.

Вместо этого мы можем установить необходимый нам провайдер. Мы будем работать с SQL Server, а для него есть провайдер **Microsoft.EntityFrameworkCore.SqlServer**. Если выбрать его, то в правой части окна в разделе **Options | Dependencies** мы увидим зависимость **Microsoft.EntityFrameworkCore.Relational**, потому что это реляционная база данных.

Взглянув на зависимости **Microsoft.EntityFrameworkCore.Relational**, мы увидим наличие зависимости от ядра **Microsoft.EntityFrameworkCore**. Получается, что если установить провайдер **SqlServer**, то он установит и пакет работы с реляционными базами, и ядро.

Нажимаем на этом провайдере кнопку **Install** и убеждаемся, что в списке есть как ядро, так и реляционный пакет (первый и последний в списке на рис. 15.2). Помимо этого, устанавливается еще много полезного и нужного для EF.

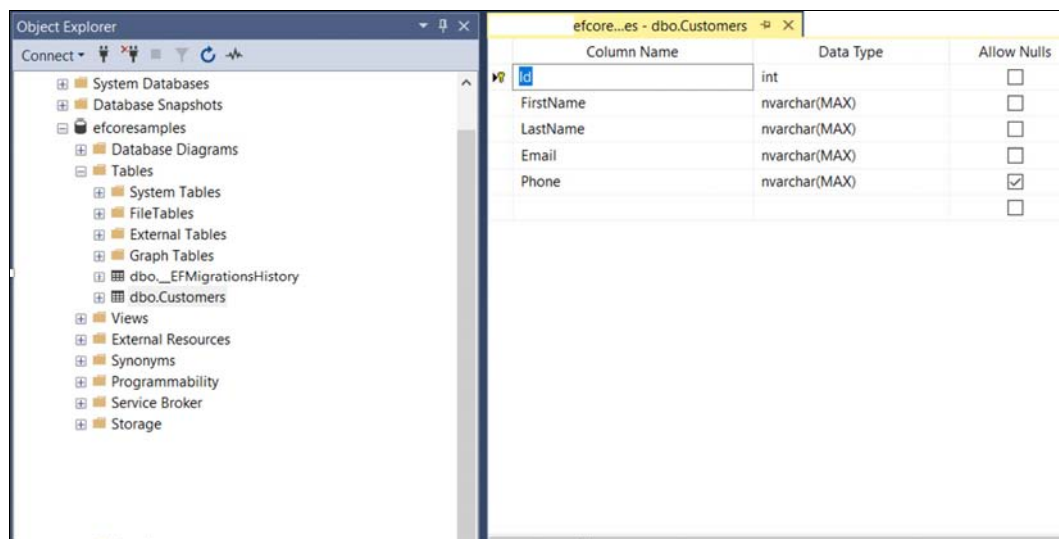


Рис. 15.2. Устанавливаемые пакеты

У меня были случаи, когда компиляция в Visual Studio завершалась ошибкой, потому что компилятор не находил пространство имен **Microsoft.EntityFrameworkCore** — в этом же случае он установит такой пакет явно.

Есть два варианта работы с базой данных — в зависимости от того, что будет первичным: код (Code First) или база данных (Database First). Если первичным выбрать код, то нужно еще установить пакет **Microsoft.EntityFrameworkCore.Design**, и именно его мы будем рассматривать в большинстве примеров. Если вы забудете сделать это, то возникнет ошибка при выполнении некоторых команд.

15.2. Код первичен

Хотя я больше люблю чистый SQL, предоставляемая EF возможность не думать о базе данных и писать только код является идеальной, когда нужно что-то быстро набросать. Я могу прибегать к подобному подходу для экспериментов или когда что-то нужно показать в коде. Я видел команды, которые использовали такой подход в качестве основного для строительства даже достаточно большого приложения. Это возможно, если вам он нравится.

Как следует из названия этого раздела, мы будем в первую очередь думать о коде и вообще не станем вспоминать о базе данных. Никакого SQL, никакого создания базы данных вручную — только код и только одна строка подключения к базе данных.

В качестве примера я решил выбрать классическую задачу электронного ведения учета заказов: счета, клиенты...

Вместо описания базы данных, мы будем описывать данные в виде кода. Давайте создадим в нашем решении папку `Models` и в ней — классы, которые будут описывать базу данных.

Почему папка должна быть именно `Models`? Можно ли использовать другое имя? Можно, и я бы даже лучше использовал имя типа `DalModels` (`Data Access Level` — модели уровня доступа к данным) или `DbModels` (модели базы данных), но почему-то я в документации и примерах чаще вижу именно `Models`, поэтому следуя за большинством и использую именно это имя.

Итак, в папке `Models` создаем файл `Custmer.cs` с одноименным классом:

```
namespace CodeFirst.Models
{
    public class Customer
    {
        public int Id { get; set; }
        public string FirstName { get; set; } = null!;
        public string LastName { get; set; } = null!;
        public string Email { get; set; } = null!;
        public string? Phone { get; set; }
    }
}
```

Здесь я создаю открытый (`public`) класс `Customer`, который описывает данные клиента. Для клиента нам понадобится:

- ❑ `Id` — числовое поле, которое впоследствии в базе данных станет первичным ключом. Нам не нужно делать никаких пометок, просто называем поле `Id` (идентификатор);
- ❑ `FirstName`, `LastName` и `Email` — имя, фамилия клиента и его E-mail (нужно же знать, с кем мы работаем);
- ❑ `Phone` — телефон, но он объявлен немного по-другому, тут нет `null!` в конце, и об этом стоит сейчас поговорить.

Почему я добавил в конце `= null!?` Дело в том, что последний .NET требует, чтобы мы явно указывали, позволяет ли поле сохранять `null` и если нет, мы его должны инициализировать. Если написать:

```
public string FirstName { get; set; }
```

то компилятор покажет ошибку: **CS8616: Non-nullable property 'FirstName' must contain a non-null value when existing constructor. Consider declaring the property as nullable.**

Смысл ошибки в том, что нет конструктора, который инициализирует значение, и нам рекомендуется указать, что можно использовать `null`.

Я не могу инициализировать эти поля — это сделает потом Entity Framework, и указывать, что то или иное поле может принимать `null`, я тоже не хочу, потому что в базе данных и в коде оно будет обязательным, — мы будем требовать от клиентов указания своего имени, фамилии и электронной почты. Выход: указать компилятору, что мы знаем о проблеме, но сейчас не будем ничего инициализировать, а клянемся, что это все же произойдет за счет добавления вот этой магии в конце: `= null!`.

Телефон тоже в реальном приложении может быть обязательным, но в нашем случае для разнообразия я решил показать, что в моем коде этот реквизит будет необязательным и позволит `null`.

Пока нам хватит одной таблицы в базе данных — давайте теперь поговорим о контексте, чтобы быстрее начать работать с данными, а потом уже вернемся и создадим еще больше классов, которые превратятся в таблицы.

15.3. Контекст базы данных

Мы описали одну таблицу, но это просто класс. И ничего не сказано о том, что этот класс каким-то образом связан с базой данных. Чтобы .NET мог связать наш класс с базой, нужно создать класс контекста базы и добавить в него `Customer`.

Давайте увидим это на практике. Создадим новый файл класса `MyStoreContext.cs`:

```
using Microsoft.EntityFrameworkCore;
using CodeFirst.Models;

namespace CodeFirst.Data
{
    internal class MyStoreContext: DbContext
    {
        public DbSet<Customer> Customers { get; set; } = null!;
    }
}
```

Класс происходит от `DbContext`. Это класс контекста базы данных, который определен в `Microsoft.EntityFrameworkCore`. Вот по этому признаку .NET сможет догадаться, что все свойства этого класса должны быть связаны с подобными в базе данных. В моем случае я добавил одно свойство `Customers` (это станет именем таблицы), которое должно иметь тип `DbSet<T>`, где `T` — это класс, который описывает данные в таблице.

А как система узнает, где находится база данных, какой сервер используется? Тут есть два варианта решения: настроить строку подключения при старте программы или переопределить метод `OnConfiguring` класса `DbContext`:

```
internal class MyStoreContext: DbContext
{
    public DbSet<Customer> Customers { get; set; } = null!;

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlServer(
            "Data Source=.;Initial Catalog=efcoresamples");
    }
}
```

Метод `OnConfiguring` получает в качестве параметра экземпляр `DbContextOptionsBuilder`, через который мы можем настроить базу. Для работы с MS SQL Server для этого вызывается метод `UseSqlServer`, которому передается строка подключения. В этой строке я подключаюсь к локальному MS SQL Server и использую базу данных с именем `efcoresamples`. Если у вас такой базы нет, то создай ее, выполнив следующий SQL-запрос:

```
create database efcoresamples;
```

Это все, что нужно знать про SQL. Вот теперь с точки зрения кода все готово, и мы можем скомпилировать проект и потом перейти в командную строку.

Чтобы выполнять команды, нужно установить инструмент `dotnet-ef`, а для этого один раз достаточно выполнить команду:

```
dotnet tool install -g dotnet-ef
```

У меня инструменты уже установлены глобально, поэтому я вижу ошибку: **Tool 'dotnet-ef' is already installed** (Инструмент `dotnet-ef` уже установлен).

Теперь можно создавать миграцию:

```
dotnet ef migrations add Initialization
```

Миграция — это специальные файлы, в которых будет находиться код создания объектов, которые мы описали в коде C#. Чуть позже мы их рассмотрим.

В команде добавления миграции последнее слово (`Initialization`) — это имя миграции. Оно может быть любым — по нему в будущем мы сможем найти сгенерированный код.

Если теперь вернуться в Visual Studio, то мы увидим, что в **Solution Explorer** появилась новая папка — **Migrations**, в которой можно увидеть два файла, и имя первого из них состоит из двух частей: текущая дата плюс имя миграции — `Initialization`, которое я указал при создании миграции. Откройте этот файл, и вы должны увидеть код, который создает таблицу `Customers`. Я приводить его здесь не буду, потому что он большой, но при этом он относительно простой, и его нетрудно прочесть.

Теоретически вы можете редактировать этот файл вручную, но я бы так делать не стал. Для нас очень важно видеть, что EF решил создать все строковые колонки, используя тип данных `nvarchar(max)`, — об этом говорит, например, вот эта строка:

```
FirstName = table.Column<string>(type: "nvarchar(max)", nullable: false)
```

Это потому, что мы нигде не указывали максимальный размер, и система не могла догадаться, какой же должен быть максимум.

На строке 24 видно, что миграция догадалась, что поле `Id` — это первичный ключ таблицы:

```
table.PrimaryKey("PK_Customers", x => x.Id);
```

Теперь наложим эти изменения на базу данных и посмотрим на результат. Выполняем в командной строке:

```
dotnet ef database update
```

И если все в порядке, то вы должны увидеть попытку собрать проект и наложить миграцию `Initialization`:

```
Build started...
Build succeeded.
Applying migration '20220609133603_Initialization'.
Done.
```

Вот как выглядит это в SQL Management Studio (рис. 15.3).

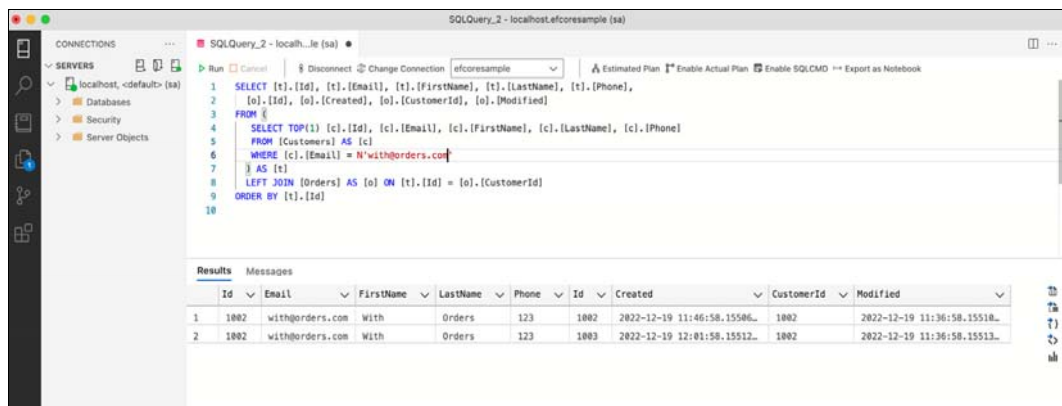


Рис. 15.3. Таблица `Customers` в SQL Management Studio

Отлично, у нас появилась таблица в базе данных, и при этом мы не делали этого вручную, т. е. не писали SQL кода, — EF Tools наложили сгенерированные на основе кода миграции на базу данных.

Из всего EF мне как раз больше всего нравится этот подход, и только из-за него я могу использовать его в каких-то прототипах.

15.4. Доступ к базе данных

Мы создали базу из одной таблицы, и чтобы завершить полный цикл, нам нужно посмотреть, как можно добавлять данные в базу и получать к ним доступ.

Сначала приведу весь код, который нужно написать в файле `Program.cs`, а потом мы поговорим про детали, почему именно так:

```
using Microsoft.EntityFrameworkCore;
using CodeFirst.Models;

namespace CodeFirst.Data
{
    using (var dbcontext = new MyStoreContext())
    {
        dbcontext.Customers.Add(
            new Customer()
            {
                FirstName = "John",
                LastName = "Doe",
                Email = "John@Doe.com"
            });

        dbcontext.Customers.Add(
            new Customer()
            {
                FirstName = "Anna",
                LastName = "",
                Email = "John@Doe.com"
            });

        dbcontext.SaveChanges();

        foreach (Customer c in dbcontext.Customers)
        {
            Console.WriteLine("First Name " + c.FirstName);
            Console.WriteLine("Last Name " + c.LastName);
        }
    }
}
```

Чтобы связать `Customer` с базой данных, в *разд. 15.3* я создал новый класс `MyStoreContext`. Раз связь уже есть, значит, мы можем использовать ее для доступа к данным. Этому классу я добавил свойство `Customers`, которое имеет тип `DbSet<Customer>`, и слово `Set` в этом случае явно намекает на то, что это может быть набором данных. Через него можно получить доступ к данным таблицы `Customers`.

Получается, что нам нужно создать сначала контекст и работать с данными в базе через него. У контекста есть строка подключения, так что дополнительные настройки не нужны, но т. к. контекст реализует интерфейс `IDisposable`, то такие классы желательно инициализировать с использованием `using`:

```
using (var dbcontext = new MyStoreContext())
```

Отлично, имея контекст, мы можем через него обращаться к данным таблицы `Customers` и добавлять в нее данные простым методом `Add`, а в качестве параметра

передавать ему экземпляр `Customer`, у которого в свойствах установлены нужные значения. Таким образом я и добавляю двух клиентов.

Но в этот момент я только добавил объекты клиентов в набор данных, но не в базу данных. Чтобы EF сохранил данные в базе, нужно вызвать метод `SaveChanges`, который есть у контекста. Именно в этот момент будет сформирован SQL-запрос для вставки данных и отправлен базе данных.

После этого можно посмотреть добавленные данные, просто перебирая все `Customers` в контексте:

```
foreach (Customer c in dbcontext.Customers)
```

Внутри цикла я просто отображаю данные текущего клиента, поэтому в результате выполнения программы я увидел в консоли:

```
First Name John  
Last Name Doe  
First Name Anna  
Last Name
```

15.5. Миграции

В *разд. 15.3* мы уже создали начальную миграцию и использовали ее для создания нашей первой таблицы в базе данных. Сейчас настало время поговорить о миграциях чуть более подробно.

Теоретически миграции можно удалять. Вы можете удалить папку `Migrations`, запустить команду создания миграций и снова получить тот же результат, но я все же не рекомендую с этим играть. Если миграция уже была наложена на базу данных, то при новой генерации могут возникнуть проблемы. Полная генерация скрипта может иметь смысл, только если вы каждый раз генерируете базу данных с нуля. Если вы изменяете существующую структуру, то лучше добавлять новые миграции и накладывать их.

Когда вы изменяете что-то в классах, которые описывают данные в базе, вам не надо генерировать все заново — вместо этого нужно добавить новую миграцию, которая изменит структуру базы так, чтобы она соответствовала вашей новой схеме.

У нас есть в базе данных клиенты, и для полноценного магазина нам необходимы еще счета и продукты. Теоретически еще нужны таблицы и для корзины, но это мы пока опустим.

Начнем с продуктов, потому что у них не будет зависимости на другие таблицы. Для этого создаем в папке `Models` новый файл `Product.cs` и в нем определяем следующий класс:

```
using System.ComponentModel.DataAnnotations;  
using System.ComponentModel.DataAnnotations.Schema;
```

```
namespace CodeFirst.Models
{
    public class Product
    {
        public int Id { get; set; }

        [Column(TypeName = "nvarchar(20)")]
        public string Name { get; set; } = null!;

        [Column(TypeName = "ntext")]
        public string Description { get; set; } = null!;

        [Column(TypeName = "decimal(6, 2)")]
        public decimal Price { get; set; }

        public DateTime Created { get; set; }

        public DateTime Modified { get; set; }
    }
}
```

Помните, что при создании таблицы клиента мы в качестве колонки для имени просто указывали тип данных `string`, и в результате был создан тип данных `nvarchar(max)`, что не очень хорошо. Хотя `nvarchar` хранит данные эффективно, все же лучше указывать более реальные значения в качестве размера строки.

И вот в этом примере перед свойством `Name` я явно указал через атрибуты, что хочу использовать тип данных `nvarchar`, и указал размерность 20 символов:

```
[Column(TypeName = "nvarchar(20)")]
```

Чтобы можно было использовать эти атрибуты, я подключил следующие пространства имен:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

Таким же образом указан тип данных `ntext` — для описания, которое может быть любой длины, и указан тип данных `decimal(6, 2)` — для цены.

Для аудита данных я указал время создания и обновления записи (это из личного опыта — по привычке я всегда создаю такие колонки. И я не один такой — в большинстве проектов я вижу подобный подход):

Следующая таблица: `OrderDetail`:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace CodeFirst.Models
{
    public class OrderDetail
```

```
{
    public int Id { get; set; }

    public int Quantity { get; set; }

    [Column(TypeName = "decimal(6, 2)")]
    public decimal Price { get; set; }

    [Column(TypeName = "decimal(6, 2)")]
    public decimal Discount { get; set; }

    public DateTime Created { get; set; }

    public DateTime Modified { get; set; }

    public int ProductId { get; set; }
    public Product Product { get; set; } = null!;

    public Order Order { get; set; } = null!;
}
```

В этой таблице будут храниться позиции заказа. Один счет может состоять из нескольких позиций. Когда вы идете в магазин и покупаете хлеб, молоко, яблоки, соки и т. д, то вам выписывают только один чек, в котором перечислены все эти позиции. Так вот, описание чека будет приведено далее, а сейчас мы смотрим на позиции в этом чеке.

Как и для любой таблицы, нам здесь нужны:

- ☐ первичный ключ;
- ☐ количество товаров в позиции (сколько хлеба или пакетов молока мы покупаем);
- ☐ цена со временем может меняться (сейчас в Канаде рекордная инфляция, и в счете нужно зафиксировать цену на момент покупки);
- ☐ скидка, если была такая;
- ☐ снова колонки аудита: создание и изменение;
- ☐ продукт `ProductId` — мы же должны знать, что куплено. Этот продукт будет приходить из таблицы `Product`. Как это узнает EF? По свойству:

```
public Product Product { get; set; } = null!;
```

- ☐ ссылка на счет, в котором будет эта строка:

```
public Order Order { get; set; } = null!;
```

Но у нас же нет таблицы `Orders`! Сейчас создадим.

Обратите внимание, что для продукта у нас есть свойство `Product` и свойство `ProductId` для него, а для свойства `Order` нет свойства `OrderId`. Почему? Для разно-

образия! Дело в том, что мы можем не указывать свойство `ProductID` явно. Мы так поступили со свойством `Product`, и EF оказался достаточно умным, чтобы понять имеющуюся связь. Поэтому, чтобы показать вам оба варианта, я для связи с таблицей `Product` добавил ключ `ProductId`, а для колонки `Order` не стал этого делать, поскольку в конце концов результат будет один и тот же.

Последняя таблица: `Order`:

```
namespace CodeFirst.Models
{
    public class Order
    {
        public int Id { get; set; }

        public int CustomerId { get; set; }
        public Customer Customer { get; set; } = null!;

        public DateTime Created { get; set; }

        public DateTime Modified { get; set; }
    }
}
```

У счета будет:

- ❑ первичный ключ;
- ❑ `CustomerId` — кто произвел покупку;
- ❑ колонки аудита: `Created/Modified`.

Мы создали описания трех новых таблиц, и их нужно добавить в наш контекст базы данных, чтобы с ними можно было потом работать:

```
internal class MyStoreContext: DbContext
{
    public DbSet<Customer> Customers { get; set; } = null!;
    public DbSet<Order> Orders { get; set; } = null!;
    public DbSet<OrderDetail> OrderDetails { get; set; } = null!;
    public DbSet<Product> Products { get; set; } = null!;
    . . .
    . . .
}
```

Отлично, мы описали необходимые объекты в виде кода и теперь готовы добавить новую миграцию, которая создаст в базе данных нужные таблицы с нужными колонками. У нас уже есть одна инициализация базы данных, которую мы выполняли, пора создать еще одну — назовем ее `OrderTables`:

```
dotnet ef migrations add OrderTables
```

В папке `Migrations` должен появиться новый файл, имя которого будет состоять из текущего времени плюс фрагмент `_OrderTables.cs`. Мы на него взглянем чуть позже.

Теперь наложим изменения на базу данных:

```
dotnet ef database update
```

В консоли должно появиться что-то типа:

```
Build started...
Build succeeded.
Applying migration '20220613184423_OrderTables'.
Done.
```

У меня уже были две миграции, но утилита наложила только одну: `OrderTables`, которая не была наложена ранее. То есть система находит изменения и накладывает только их.

Запустите SQL Management Studio и убедитесь, что изменения наложены на базу данных. Теперь в базе должны быть таблицы: `Customers`, `Orders`, `OrderDetails`, `Customers`. Помимо этого, есть еще одна таблица: `[_EFMigrationsHistory]`, в которой хранятся все SQL-файлы, которые уже были наложены на базу данных. Посмотрим на эту таблицу:

```
select * from [dbo].[_EFMigrationsHistory]
```

Этот запрос должен показать следующие строки:

MigrationId	ProductVersion
20220613193039_Initialization	6.0.5
20220613193115_OrderTables	6.0.5

Здесь у нас две колонки: идентификатор миграции и версия продукта. По этой таблице утилиты EF будут узнавать, что уже наложено на базу данных, а что нет и что нужно наложить.

Ок, понятно. Каждый раз, когда мы изменяем структуру данных в `C#`-файлах, мы можем создавать новые и новые миграции и потом накладывать эти изменения на базу данных.

Посмотрим, на сгенерированную миграцию `OrderTables` — там есть кое-что интересное:

```
migrationBuilder.CreateTable(
    name: "Products",
    columns: table => new
    {
        Id = table.Column<int>(type: "int", nullable: false)
            .Annotation("SqlServer:Identity", "1, 1"),
        Name = table.Column<string>(type: "nvarchar(20)", nullable: false),
        Description = table.Column<string>(type: "ntext", nullable: false),
        Price = table.Column<decimal>(type: "decimal(6,2)", nullable: false),
        Created = table.Column<DateTime>(type: "datetime2", nullable: false),
        Modified = table.Column<DateTime>(type: "datetime2", nullable: false)
    },
migrationBuilder.CreateTable(
    name: "OrderDetails",
```

```

columns: table => new
{
    Id = table.Column<int>(type: "int", nullable: false)
        .Annotation("SqlServer:Identity", "1, 1"),
    Quantity = table.Column<int>(type: "int", nullable: false),
    Price = table.Column<decimal>(type: "decimal(6,2)", nullable: false),
    Discount = table.Column<decimal>(type: "decimal(6,2)",
        nullable: false),
    Created = table.Column<DateTime>(type: "datetime2", nullable: false),
    Modified = table.Column<DateTime>(type: "datetime2", nullable: false),
    ProductId = table.Column<int>(type: "int", nullable: false),
    OrderId = table.Column<int>(type: "int", nullable: false)
},

```

Весь файл я приводить не стал, а показал только описание двух таблиц. В них я выделил строки в коде создания таблицы `OrderDetails`, где вы можете видеть, что EF сгенерировал код для создания `ProductId` и `OrderId`, хотя `OrderId` я в своем коде не указывал.

Еще одной интересной особенностью нового скрипта является наведение связей внешних ключей:

```
table.ForeignKey
```

Опять же, фреймворк оказался настолько умным, что по наличию свойств типа:

```
public Product Product { get; set; } = null!
```

были созданы внешние ключи для связи таблицы `OrderDetail` с продуктом `Product` и `Order`, а также связь между `Order` и `Customer`.

Ну и напоследок EF создал еще и индексы для всех внешних ключей. Например, следующий код создал индекс на таблицу `OrderDetail` — колонку `OrderId`:

```

migrationBuilder.CreateIndex(
    name: "IX_OrderDetails_OrderId",
    table: "OrderDetails",
    column: "OrderId");

```

MS SQL Server не создает автоматически индексы для колонок, которые участвуют в связях внешнего ключа, а это может быть плохо с точки зрения производительности. Да, бывают редкие случаи, когда такой индекс не нужен, но все же в большинстве случаев они нужны, и поэтому EF играет тут с точки зрения обеспечения надежности и на всякий случай создает индекс.

Если вы опытный пользователь MS SQL Server, то можете в Management Studio посмотреть сгенерированные таблицы и убедиться, что типы данных соответствуют тому, что мы указывали, и есть внешние ключи.

Если вы еще не накладывали миграцию на базу данных, то ее можно удалять и можно даже изменять файл кода, чтобы он лучше соответствовал тому, что вы хотите получить. Но если файл уже наложен, то лучше его не изменять.

15.6. База данных первична

Когда вы работаете над проектом, где уже есть база данных, и ее нужно использовать с EF, или если вы хотите иметь чуть больше контроля над тем, как выполняются миграции, и не доверяете утилитам EF, то можно сначала создать нужную вам структуру базы данных, а потом сгенерировать код для этой структуры.

Теоретически можно все писать вручную: код на C# и SQL-скрипты для обновления базы данных, но это двойная работа. Лучше все же выбрать, что вы будете писать вручную, а что будет генерироваться.

В предыдущем разделе мы сгенерировали базу данных, но давайте представим, что она была создана вручную. Впрочем, если вы хотите, чтобы ваш тест был чистым, то можете создать новую базу данных вручную и тестировать не ней все, что я буду сейчас рассматривать.

Давайте создадим новый проект (я назвал его DbFirst), который снова будет консолью. Также добавим пакеты для работы с EF: `Microsoft.EntityFrameworkCore.SqlServer` и `Microsoft.EntityFrameworkCore.Tools`.

Теперь в командной строке выполним следующую команду:

```
dotnet ef dbcontext scaffold "Data Source=.;Initial Catalog=efcoresamples;
Trusted_Connection=True" Microsoft.EntityFrameworkCore.
                        SqlServer --context-dir Data --output-dir Models
```

Рассмотрим эту большую команду по частям:

- ❑ `dotnet ef dbcontext scaffold` — команда, которая по факту просто говорит, что мы хотим сгенерировать файлы контекста базы данных. После этого должна идти строка подключения и имя провайдера подключения к базе данных;
- ❑ `"Data Source=.;Initial Catalog=efcoresamples;Trusted_Connection=True"` — строка подключения заключена в двойные кавычки, потому что в ней могут быть символы, которые имеют для командной строки специальное значение;
- ❑ `Microsoft.EntityFrameworkCore.SqlServer` — провайдер, который будет использоваться для генерации кода. В нашем случае это `SqlServer`.

Остальные параметры необязательные, но если их не указать, то все файлы будут созданы в текущей папке, и это будет большая куча-мала, а я предпочитаю все раскладывать по полочкам:

- ❑ `--context-dir Data` — в папку `Data` попадет файл контекста базы данных;
- ❑ `--output-dir Models` — в папке `Models` будут храниться файлы моделей.
- ❑ Выполнив эту команду, вы увидите две папки: `Data` и `Model`, в которых будут содержаться файлы, очень похожие по своей структуре на те, что мы писали вручную в предыдущих разделах.

Посмотрим на одну из моделей, что сгенерирована для таблицы `Product`:

```
public partial class Product
{
    public Product()
```

```

{
    OrderDetails = new HashSet<OrderDetail>();
}

public int Id { get; set; }
public string Name { get; set; } = null!;
public string Description { get; set; } = null!;
public decimal Price { get; set; }
public DateTime Created { get; set; }
public DateTime Modified { get; set; }

public virtual ICollection<OrderDetail> OrderDetails { get; set; }
}

```

Первое, на что мы обращаем внимание: класс объявлен как `partial`. Это не случайно. Мы не должны редактировать и изменять сгенерированные файлы вручную, чтобы можно было в любой момент удалить их и сгенерировать заново, когда изменится структура базы данных. Тогда как расширять возможности сгенерированных классов? Простое решение: создавать `partial`-классы, и EF уже сделал необходимое.

Вторая разница: где описание размера полей? Когда мы работали вручную, то я указывал атрибуты для размера строк и для размерности `decimal`, — куда это делось? На самом деле эта информация больше нужна для генерации базы данных, а не для C#-кода, ведь тип `string` с атрибутом базы данных в 20 символов не будет никак ограничивать нас.

Но ограничения размера и индексы все же были для нас сгенерированы — они просто находятся в контексте. Взгляните на файл контекста `efcoresamples Context.cs`, где появился еще один метод:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Order>(entity =>
    {
        entity.HasIndex(e => e.CustomerId, "IX_Orders_CustomerId");
        entity.HasOne(d => d.Customer)
            .WithMany(p => p.Orders)
            .HasForeignKey(d => d.CustomerId);
    });

    modelBuilder.Entity<OrderDetail>(entity =>
    {
        entity.HasIndex(e => e.OrderId, "IX_OrderDetails_OrderId");
        entity.HasIndex(e => e.ProductId, "IX_OrderDetails_ProductId");
        entity.Property(e => e.Discount).HasColumnType("decimal(6, 2)");
        entity.Property(e => e.Price).HasColumnType("decimal(6, 2)");
    });
}

```



```
entity.HasOne(d => d.Order)
    .WithMany(p => p.OrderDetails)
    .HasForeignKey(d => d.OrderId);

entity.HasOne(d => d.Product)
    .WithMany(p => p.OrderDetails)
    .HasForeignKey(d => d.ProductId);
});

modelBuilder.Entity<Product>(entity =>
{
    entity.Property(e => e.Description).HasColumnType("ntext");
    entity.Property(e => e.Name).HasMaxLength(20);
    entity.Property(e => e.Price).HasColumnType("decimal(6, 2)");
});

OnModelCreatingPartial(modelBuilder);
}
```

На мой взгляд, это менее читаемый подход, потому что описание типов данных находится в отдельном файле, и читабельность получается не самая лучшая. Я считаю, что лучше использовать атрибуты — тогда поля и их описания будут находиться в одном файле и классе.

Теперь мы можем в файле Program.cs написать абсолютно такой же код, который был в предыдущем примере, чтобы увидеть данные, — только нужно заменить имя контекста (сгенерированное имя соответствует имени базы данных) и использовать новые пространства имен.

Так что работа с подходом, где первична база данных, не сильно отличается от подхода, когда первичен код. С точки зрения C# все выглядит идентично — разница только в командах, которые генерируют код или генерируют миграции и накладывают их на базу данных.

Так что если база данных первична, то вы сами пишете SQL для изменения базы данных и генерируете только C#-код, а миграции EF не нужны.

15.7. LINQ

Мы научились создавать модели и рассмотрели простой пример, в котором в базу данных добавлялись две записи клиентов и отображалось все их содержимое. Но если запустить этот пример еще раз, то в базе данных появятся дубликаты, а при третьем запуске у нас уже будет по три копии одних и тех же записей.

Чтобы этого не происходило, перед добавлением можно проверять, нет ли уже в базе данных нужной записи. И вот тут прекрасно подходит LINQ.

В первом примере мы просто перебирали все записи клиентов `dbcontext.Customers` в цикле, но благодаря EF мы можем искать записи с помощью LINQ. Следующая

строка ищет в базе запись по Email, и если пользователь John@Doe.com существует в базе, то мы увидим в результате объект, иначе будет null:

```
Customer? johnCustomer = dbcontext.Customers.Where(
    m => m.Email == "John@Doe.com").FirstOrDefault();
```

То есть так мы можем сделать проверку до добавления и защититься от дубликатов:

```
Customer? johnCustomer = dbcontext.Customers.Where(
    m => m.Email == "John@Doe.com").FirstOrDefault();
if (johnCustomer == null)
{
    johnCustomer = new Customer()
    {
        FirstName = "John",
        LastName = "Doe",
        Email = "John@Doe.com"
    };
    dbcontext.Customers.Add(johnCustomer);
}
```

В своих собственных проектах я пока не использую и не планирую использовать LINQ, но если в компаниях, где я работаю, программисты его используют, то я вынужден согласиться с этим и использую, но не по своей воле...

15.8. Связи: один ко многим

Мы уже создали структуру небольшой базы данных и даже написали пример, который работает только с одной таблицей. Но в реальной жизни не так часто приходится работать с одной таблицей — обычно нужно связывать различные таблицы.

В моделях связи между сущностями (можно говорить — таблицами) уже наведены, поскольку каждая модель у нас представляет в базе данных таблицу.

Когда мы делали модель счета Order, то там уже навели связь между таблицами, но не использовали ее:

```
public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public Customer Customer { get; set; } = null!;
    public DateTime Created { get; set; }
    public DateTime Modified { get; set; }
}
```

У модели Order имеется свойство Customer, и это как раз и есть внешний ключ на таблицу/модель Customer. Давайте еще раз повторим теорию наведения связей между таблицами на примере таблицы OrderDetail, у которой тоже должна быть связь со счетом и продуктом.

Вспомним, как была объявлена таблица `OrderDetail`:

```
public class OrderDetail
{
    . . .
    public int ProductId { get; set; }
    public Product Product { get; set; } = null!;

    public Order Order { get; set; } = null!;
}
```

Я здесь убрал часть полей и оставил наиболее интересные, которые мы сейчас и рассмотрим.

А самое интересное тут хранится в свойствах: `Product` и `ProductId`. Свойство `Product` является объектом другой модели, и это как раз и есть связь между счетами и продуктом. Этим мы указываем связь на таблицу `Product`.

Указание свойства `ProductId` необязательно. Обратите внимание: при указании следующей связи: `Order` — я не указываю свойства `OrderId`, но если посмотреть на базу данных, то в ней вы увидите поле `OrderId`, которое является внешним ключом. Так стоит ли указывать явно колонки `ProductId` и `OrderId`? Если вы будете работать с ними в будущем, то можно — просто для удобства.

Имя колонки для связи (`ProductId`) неслучайно — это правило, которому вы должны следовать для указания имени колонки внешнего ключа: имя модели, после которого должно идти `Id`. Если указать имя иначе, не следуя этому правилу, то миграция может не понять ваших намерений и создать отдельную колонку в таблице, которая не будет являться внешним ключом.

Таким способом мы наводим классическую связь «один ко многим» — один счет `Order` может содержать несколько позиций `OrderDetails`.

Мы можем отразить это прямо в модели:

```
public class Order
{
    public int Id { get; set; }
    public int CustomerId { get; set; }
    public Customer Customer { get; set; } = null!;
    public DateTime Created { get; set; }
    public DateTime Modified { get; set; }

    public List<OrderDetail> OrderDetails { get; set; }
}
```

Здесь я добавил модели `Order` новое свойство — список из `OrderDetails`. Тут можно использовать не только списки `List`, но и `IEnumerable` или `Collection` — в зависимости от того, чего вы хотите добиться, но об этом мы поговорим позже.

При наведении всех связей я объявляю поля для внешних ключей как целочисленные и не поддерживающие `NULL`. У таблицы счетов есть связь на клиента:

```
public int CustomerId { get; set; }
```

Это не nullable число, потому что создать счет и не указывать клиента, который его создал, будет немного странно. В отношении «один ко многим» внешние ключи обычно не нулевые. Но если есть ситуация, когда существует анонимный счет и его могут создать без регистрации на сайте, то мы можем явно указать, что клиент может отсутствовать:

```
public int? CustomerId { get; set; }
```

В этом случае мы сможем создавать счета без указания клиента.

15.9. Работа со связями

Мы знаем, как добавлять данные в одну таблицу, а сейчас разберемся, как добавлять их в связанные таблицы.

Сначала нужно найти запись клиента, для которой мы будем добавлять счет. Ну а потом в список `Order` добавляем новый экземпляр счета, и вот тут как раз происходит магия — мы просто задаем в свойстве `CustomerId` идентификатор найденного клиента:

```
Customer? johnCustomer = dbcontext.Customers.Where(
    m => m.Email == "John@Doe.com").FirstOrDefault();
dbcontext.Orders.Add(
    new Order()
    {
        CustomerId = johnCustomer.Id,
        Created = DateTime.Now,
        Modified = DateTime.Now
    });

dbcontext.SaveChanges();
```

Отлично, мы смогли сохранить данные нового счета в базе данных, а теперь давайте посмотрим, как можно прочитать эти данные.

Первый способ — это пролистать все счета, к которым мы можем получить доступ через `dbcontext.Orders`:

```
foreach (Order order in dbcontext.Orders)
{
    Console.WriteLine("First Name " + order.Customer.FirstName);
    Console.WriteLine("Last Name " + order.Customer.LastName);
    Console.WriteLine("Order Id " + order.Id);
    Console.WriteLine("Order Date " + order.Created.ToString());
}
```

Раз мы можем прочитать `dbcontext.Orders`, то, возможно, можем задействовать LINQ — чтобы фильтровать данные и посмотреть только определенные счета? Да, мы можем делать так:

```
foreach (Order order in
    dbcontext.Orders.Where(o => o.CustomerId == johnCustomer.Id)) {
    Console.WriteLine("First Name " + order.Customer.FirstName);
    Console.WriteLine("Last Name " + order.Customer.LastName);
    Console.WriteLine("Order Id " + order.Id);
    Console.WriteLine("Order Date " + order.Created.ToString());
}
```

Здесь я использую LINQ, чтобы увидеть только счета, которые принадлежат Джону (john):

```
dbcontext.Orders.Where(o => o.CustomerId == johnCustomer.Id)
```

Третий способ предоставляет нам определенную гибкость, но потребует некоторых телодвижений. У класса `Order` я уже добавил в модели связь с `OrderDetails`, то же самое можно сделать и для модели клиентов:

```
public class Customer
{
    public int Id { get; set; }
    public string FirstName { get; set; } = null!;
    public string LastName { get; set; } = null!;
    public string Email { get; set; } = null!;
    public string? Phone { get; set; }
    public List<Order> Orders { get; set; }
}
```

Последнее свойство — это список, через который мы можем узнать счета, которые принадлежат текущему клиенту. Вы, наверное, уже подумали, что `johnCustomer.Orders` магическим образом заработал? Но нет — если попробовать обратиться к этому свойству, то получится, что вы обращаетесь к свойству `NULL`, и произойдет исключительная ситуация, потому что по умолчанию EF не будет ничего грузить, — это сделано в целях оптимизации. Мы должны явно указать, что хотим подгружать счета.

Так что сначала — чтобы получить доступ к расширению `Include` — подключим пространство имен:

```
using Microsoft.EntityFrameworkCore;
```

Теперь при загрузке данных клиента мы должны с помощью `Include` явно указать, что хотим видеть и счета тоже:

```
Customer? johnCustomer = dbcontext.Customers
    .Where(m => m.Email == John@Doe.com)
    .Include(m => m.Orders)
    .FirstOrDefault();
```

Если не вызвать `Include`, то свойство `Orders` окажется нулевым, и ничего не будет загружено. Как я сказал, это сделано с целью оптимизации. Если вы действительно хотите иметь доступ к счетам, то указываете `Include`, иначе не стоит тратить на это ресурсы.

Вот теперь свойство `johnCustomer.Orders` не будет нулевым, и в него будут загружены счета пользователя Джона:

```
foreach (Order order in johnCustomer.Orders)
{
    Console.WriteLine("First Name " + order.Customer.FirstName);
    Console.WriteLine("Last Name " + order.Customer.LastName);
    Console.WriteLine("Order Id " + order.Id);
    Console.WriteLine("Order Date " + order.Created.ToString());
}
```

15.10. Связи: многие ко многим

До сих пор все данные были прямолинейными: у одного клиента могло быть несколько счетов, но при этом счет мог принадлежать только одному клиенту. Но есть чуть более сложная задача, когда связь происходит по схеме «многие ко многим».

Чтобы проще объяснить суть подхода «многие ко многим», я создам отдельный проект `ClassesManyToMany`, и мы рассмотрим эту проблему на классической задаче — студенты и курсы: один студент может подписаться на много курсов, много студентов могут подписаться на один и тот же курс.

При работе с чистым SQL — если понять идею — эта задача решается относительно просто: нужно добавить связующую таблицу, через которую и будет организовываться связь.

Раньше в EF связь «многие ко многим» реализовывалась не так просто, но потом Microsoft решила упростить жизнь программистам и изменила подход. Так что мы здесь рассмотрим современный подход, который очень прост.

Нам понадобятся классы — студенты и предмет, который студент хочет изучать:

```
public class Subject
{
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Student
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Теперь, кому из них добавить список? Если мы добавим его студенту:

```
public class Subject
{
    public int Id { get; set; }
    public string Name { get; set; }
```

```
        public string StudentId { get; set; }
    }
    public class Student
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public List<Subject> Subjects { get; set; } = null!;
    }
```

то у него будет связь на предмет, и мы сможем назначить Васе несколько предметов: математику, русский, литературу и т. д. Потом создать Петю, и ему тоже назначить математику и литературу, но это уже будут другие математика и литература, потому что в базе данных у `Subject` может быть только один `StudentId`.

Как уже отмечалось, при работе с чистым SQL в случае с подходом «многие ко многим» нужно создавать в базе данных дополнительную таблицу, которая будет хранить связи. Неужели в EF мы тоже должны создать модель связующей таблицы для наведения связи?

Нет, в .NET EF Core, начиная с 5-й версии, изменили подход к созданию связей, и теперь все стало намного проще — никаких ключей `Id` для связи добавлять не нужно, только списки:

```
public class Student
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Subject> Subjects { get; set; } = null!;
}

public class Subject
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Student> Students { get; set; } = null!;
}
```

Здесь добавлены только списки — ни у одной модели нет `StudentId` и нет `SubjectId`, которые как раз задавали бы единственную связь. Нет никаких связующих таблиц, все достаточно просто: у студента может быть несколько предметов, у предмета может быть несколько студентов.

Теперь нам нужен контекст:

```
internal class SchoolContext : DbContext
{
    public DbSet<Student> Students { get; set; } = null!;
    public DbSet<Subject> Subjects { get; set; } = null!;
```

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    if (optionsBuilder.IsConfigured)
        return;

    optionsBuilder.UseSqlServer(
        "Data Source=.;Initial Catalog=samples;Trusted_Connection=True;
    ");
}
```

Компилируем проект и можно сгенерировать миграцию:

```
dotnet ef migrations add initial
```

Взгляните на содержимое файла `SchoolContextModelSnapshot.cs`, который был в результате создан. Вы должны в нем увидеть создание таблицы студентов:

```
modelBuilder.Entity("ClassesManyToMany.Models.Student", b =>
```

Здесь будет таблица предметов:

```
modelBuilder.Entity("ClassesManyToMany.Models.Subject", b =>
```

а также связующая таблица студентов и предметов:

```
modelBuilder.Entity("StudentSubject", b =>
```

Хотя мы явно такую модель в коде не объявляли, но, увидев, что студенты и предметы ссылаются друг на друга списками, EF догадался, что нам нужна связь «многие ко многим» и добавил код для создания этой таблицы.

Давайте наложим изменения на базу данных:

```
dotnet ef database update
```

и проверим результат в базе данных с помощью SQL Management Studio. Мы увидим там три таблицы: `Students`, `StudentSubject` и `Subjects` со всеми колонками, которые мы ожидали бы увидеть в случае с работой в чистом SQL (рис. 15.4). Обратите внимание на таблицу `StudentSubject` — тут составной первичный ключ из обеих колонок, и каждая из колонок в отдельности является внешним ключом.

В старых версиях .NET Core вы должны были бы описывать промежуточную таблицу вручную, и если вам доведется работать в компании, где все еще застряли в старой версии EF Core, то первый вопрос, который вы должны задать: почему они не обновляются на .NET 6 или более новый? Если программисты вынуждены использовать .NET Framework, то у них могут быть причины для того, чтобы работать со старым EF, но EF Core — это из мира .NET Core, который уже не так сложно обновлять на современные версии — такие как .NET 5 или 6.

Иногда бывает необходимо самостоятельно добавить связь просто потому, что нужно отслеживать какую-то дополнительную информацию — например, дату, когда студент записался на курс. В этом случае мы можем создать модель в виде:

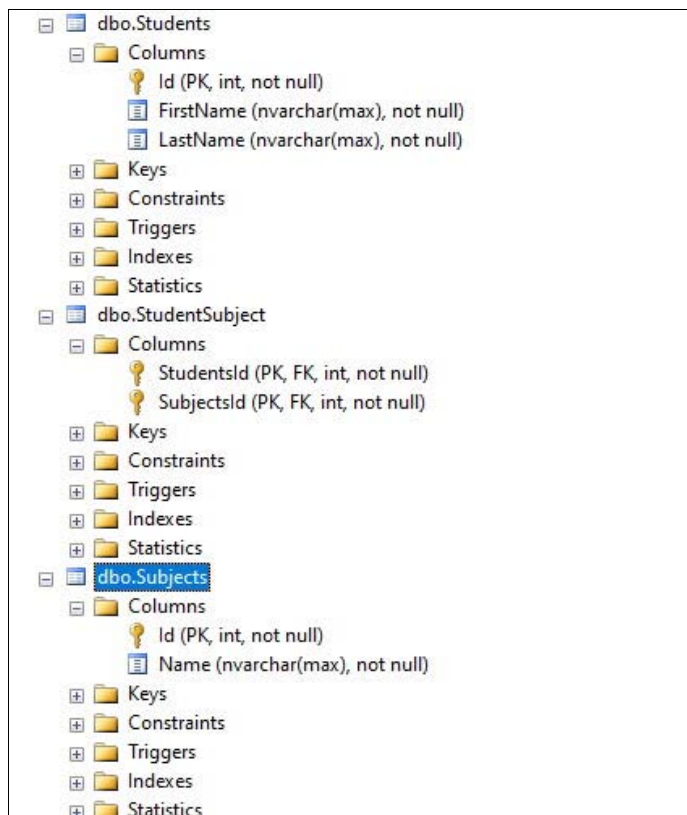


Рис. 15.4. Таблицы в базе данных

```
public class StudentSubject
{
    public int StudentId { get; set; }
    public int SubjectId { get; set; }
    public DateTime Enrolled { get; set; }
    public int Status { get; set; }
}
```

Все остальное остается без изменений — мы просто описали модель `StudentSubject` вручную. И здесь, помимо двух колонок: `StudentId` и `SubjectId`, я добавил дополнительную информацию: когда студент записался на класс и его статус.

Теперь идем в контекст `SchoolContext.cs` и добавляем переопределение метода `OnModelCreating`, где мы можем настроить связь:

```
internal class SchoolContext : DbContext
{
    // Здесь у нас объявлены DbSet
    public DbSet<Student> Students { get; set; } = null!;
    public DbSet<Subject> Subjects { get; set; } = null!;
    public DbSet<StudentSubject> StudentSubject { get; set; } = null!;
```

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    // Тут у нас настраивается соединение с БД
}

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
        .HasMany(m => m.Subjects)
        .WithMany(m => m.Students)
        .UsingEntity<StudentSubject>(
            m => m.HasOne<Subject>().WithMany(),
            m => m.HasOne<Student>().WithMany()
        );
}
}
```

В методе `OnModelCreating` мы настраиваем связь и отмечаем, что наша модель `Student` имеет связь «многие ко многим» с моделью `Subjects`:

```
modelBuilder.Entity<Student>()
    .HasMany(m => m.Subjects)
    .WithMany(m => m.Students)
```

Далее мы говорим, что связь происходит через `StudentSubject`:

```
.UsingEntity<StudentSubject>(
    m => m.HasOne<Subject>().WithMany(),
    m => m.HasOne<Student>().WithMany()
)
```

Этого достаточно, но мы можем пойти дальше и сделать так, что колонка `Enrolled` будет заполняться автоматически. Это дата создания записи, а в SQL можно задавать для колонки значения по умолчанию и сказать, что при создании записи в колонку будет попадать определенное значение. В нашем случае мы хотим, чтобы при создании записи в `Enrolled` попадала текущая дата, и это можно сделать прямо из кода:

```
modelBuilder.Entity<Student>()
    .HasMany(m => m.Subjects)
    .WithMany(m => m.Students)
    .UsingEntity<StudentSubject>(
        m => m.HasOne<Subject>().WithMany(),
        m => m.HasOne<Student>().WithMany()
    )
    .Property(m => m.Enrolled)
    .HasDefaultValueSql("getdate()");
```

Я добавил две строки, которые прямо дословно, на английском, говорят, что для свойства `Enrolled` мы хотим задать значение по умолчанию в виде SQL-команды `getdate()`.

Создайте новую миграцию:

```
dotnet ef migrations add manual-many-to-many
```

Наложите ее на базу данных:

```
dotnet ef database update
```

И теперь, посмотрев на базу из SQL Management Studio, убеждаемся, что миграция добавила нужные колонки и значение по умолчанию таблице связи «многие ко многим» (рис. 15.5).

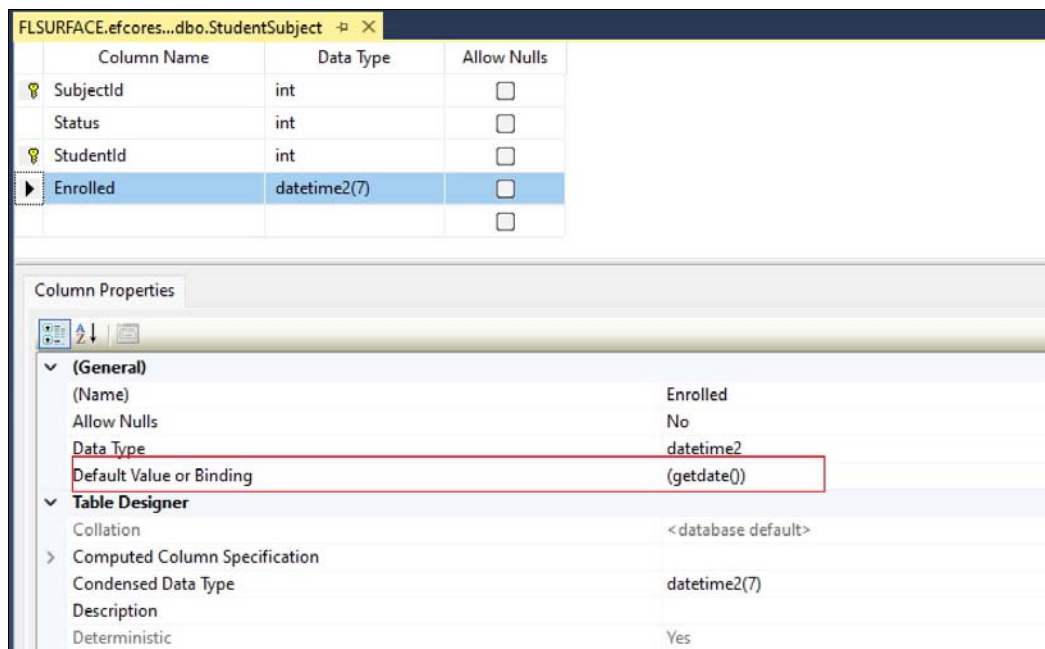


Рис. 15.5. Таблицы в базе данных

Лично мне кажется, что код в C# — не самый простой для чтения, но ведь для подобной задачи не так-то просто придумать синтаксис проще. Я думаю, что со временем все же можно привыкнуть к этому синтаксису.

15.11. Связи: один к одному

Я расположил в этой главе описания связей по мере популярности из своего личного опыта. Чаще всего я работаю со связью «один ко многим». Чуть реже — «многие ко многим». Ну и самое редкое — «один к одному». В последних компаниях, где я работал, у меня было два крупных случая связи «один к одному». Случались еще менее яркие, но вот крупных было только два. Так, при работе с Sony мне пришлось работать с двумя таблицами:

- ❑ **Member** — базовая информация о пользователе. Минимум необходимых данных, которые были необходимы при отображении пользователю;

❑ `MemberDetails` — дополнительная информация, которая нужна была редко, и ее вынесли в отдельную таблицу просто для того, чтобы `Member` была не такой большой. С точки зрения производительности это, действительно, может быть неплохо, если часто приходится сканировать все данные.

У вас есть идеи, как можно сделать связь «один к одному»? Давайте создадим новый класс `StudentData`:

```
public class StudentData
{
    public int Id { get; set; }
    public int StudentId { get; set; }

    public string Address { get; set; }
    public string CityName { get; set; }
}
```

У `StudentData` есть свой ключ и связь на студента.

Теперь у студента можно добавить связь на `StudentData`:

```
public class Student
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public List<Subject> Subjects { get; set; } = null!;
    public StudentData StudentData { get; set; }
}
```

В отличие от связи «многие ко многим» или «один ко многим», свойство `StudentData` не является списком — это просто переменная, потому что мы не ожидаем множества значения, а только одно.

В принципе, это всё. Такой подход можно часто увидеть в Интернете, но это не самый эффективный вариант, потому что на самом деле создается связь «один ко многим» — просто в таблице `Student` мы ограничиваемся только одной записью.

У студента есть ключ `Id`, и у `StudentData` есть такой же, и они должны быть равны. Только при их равенстве мы точно сможем гарантировать, что связь идет «один к одному».

Давайте исправим эту двусмысленность и сделаем что-то более красивое:

```
public class StudentData
{
    public int Id { get; set; }
    public string Address { get; set; }
    public string CityName { get; set; }
    public Student Student { get; set; }
}
```

Я здесь убрал `StudentId` и вместо этого использую объект для связи.

Теперь идем в контекст и в методе `OnModelCreating` указываем связь вручную:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Student>()
        .HasOne(sd => sd.StudentData)
        .WithOne(s => s.Student)
        .HasForeignKey<StudentData>(s => s.Id);
}
```

Здесь я говорю, что у модели `Student` есть одна `StudentData`, которая ссылается на нашего одного студента, и добавляю внешний ключ на `Id`.

Создайте миграцию и посмотрите, что будет для нас сгенерировано. Можно наложить миграцию на базу данных и убедиться, что у `StudentData` нет дополнительных колонок для связи, а есть только `Id`, который является первичным ключом и одновременно внешне связывается со `Student`.

Вот это уже чистый и классический подход «один к одному», и я бы рекомендовал использовать именно его.

15.12. Журналы

Работая под Windows, вы можете воспользоваться очень удобной программой SQL Server Profiler, которая способна подключиться к определенному серверу базы данных и начать перехватывать запросы. Под macOS я пока с такой программой не сталкивался, поэтому очень хотелось бы иметь возможность посмотреть, какие запросы выполняет EF, чтобы превратить все наши изменения в виде кода в реальные изменения в базе данных.

Для этого в EF можно использовать журналы, работу с которыми обеспечивает компонент `LoggerFactory`. В окне Проводника решения щелкните правой кнопкой мыши на **Dependencies** (Зависимости) и выберите **Manage NuGet Packages** (Управление пакетами NuGet) — в основной области среды разработки должна появиться вкладка **Browse**. Перейдите на нее и в строку поиска введите `LoggerFactory`. В результате должен быть найден пакет `Microsoft.Extensions.Logging.Abstractions` (рис. 15.6). Если так его найти не удастся, то укажите в строке поиска его полное имя.

Затем установите пакет `Microsoft.Extensions.Logging.Console`, потому что мы будем отображать данные пока прямо в консоли и воспользуемся готовым решением.

Теперь перейдите в файл `MyStoreContext.cs` и в `OnConfiguring` добавьте выделенную полужирным строку:

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    if (optionsBuilder.IsConfigured)
        return;
```

```

optionsBuilder
    .UseLoggerFactory(
        LoggerFactory.Create(builder => { builder.AddConsole(); })
    )
    .UseSqlServer("Строка подключения");
}

```

Выделенные строки включают журналирование и в качестве средства журналирования добавляется консоль.

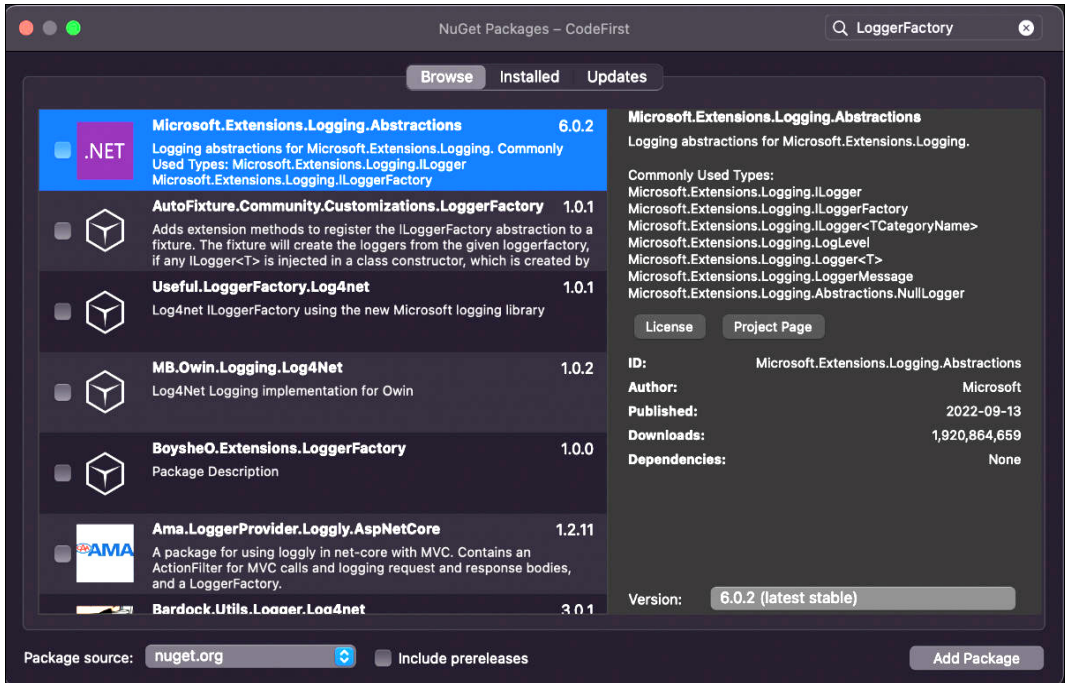


Рис. 15.6. Установка LoggerFactory

Чтобы скомпилировать проект, нужно добавить пространство имен:

```
using Microsoft.Extensions.Logging;
```

Запускаем приложение, и теперь в консоли будет виден не только вывод информации, которую мы просили отображать на экране, но еще и SQL-запросы, которые выполнялись к базе данных. Вот результат одного из примеров, который я рассматривал раньше, когда мы говорили об опасных связях:

```

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options:
      None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (18ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']

```

```

SELECT [t].[Id], [t].[Email], [t].[FirstName], [t].[LastName],
[t].[Phone], [o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified]
FROM (
    SELECT TOP(1) [c].[Id], [c].[Email], [c].[FirstName],
    [c].[LastName], [c].[Phone]
    FROM [Customers] AS [c]
    WHERE [c].[Email] = N'John@Doe.com'
) AS [t]
LEFT JOIN [Orders] AS [o] ON [t].[Id] = [o].[CustomerId]
ORDER BY [t].[Id]

```

Таким образом, получается, что EF — всего лишь обертка, которая абстрагирует нас от SQL, а общение с базой данных все еще продолжается на классическом языке запросов SQL? Да, поэтому все, кто считает, что EF способен получать данные из базы быстрее, чем чистый SQL, немного заблуждаются, потому что для доступа к данным используется всё тот же SQL, и тут нет никакой магии. Ваш чистый SQL будет медленнее только в одном случае — когда он написан плохо. Так что если EF генерирует запросы лучше, чем их пишет программист вручную, то да — EF будет быстрее.

Но в случае с C#-приложением не все завязано только на запросы, есть еще вопрос маппинга данных из результата запроса в C#-объекты, и тут EF в последнее время получил великолепные улучшения, и уже не находится позади всех, а постепенно пробивается в лидеры. Основным лидером пока еще остается Dapper ORM, но EF в версии Core приближается к нему очень быстро. EF версии Framework первых выпусков был очень медленным и при этом генерировал ужасные SQL-запросы, которые тяжело было читать, а это необходимо при отладке или при работе над оптимизацией.

Но вернемся к журналам. Глядя на то, сколько информации появилось в консоли, может показаться, что найти нужный запрос будет невозможно. Да, поэтому у EF есть очень удобная возможность — теги для запросов. Например, мы можем добавить специальный тег с помощью метода `TagWith` и в качестве параметра передать текст с описанием:

```

Customer? johnCustomer =
    dbcontext.Customers.TagWith("Getting all customners")
        .Where(m => m.Email == "John@Doe.com")
        .Include(m => m.Orders)
        .FirstOrDefault();

```

Теперь в журнале перед самым запросом появится тег в виде комментария:

```

info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (44ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']
-- Getting all customners

SELECT [t].[Id], [t].[Email], [t].[FirstName], [t].[LastName],
[t].[Phone], [o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified]
FROM (

```

Применяя этот способ, найти нужный запрос будет намного проще.

Есть еще один вариант отображать информацию. В контексте базы можно заметить:

```
.UseLoggerFactory(
    LoggerFactory.Create(builder => { builder.AddConsole(); })
)
```

На:

```
.LogTo(log => Console.WriteLine(log));
```

Этот метод предоставляет по умолчанию очень много подробной информации — еще больше, чем мы видели при использовании `UseLoggerFactory`. Но количество информации можно контролировать, указав дополнительный параметр `LogLevel`:

```
.LogTo(log => Console.WriteLine(log), LogLevel.Debug)
```

15.13. Как сохраняются данные?

Получив доступ к журналам, мы смогли увидеть, что EF лишь использует SQL. А давайте попробуем создать новый пример, который будет просто добавлять две новые записи в базу данных:

```
using (var dbcontext = new MyStoreContext())
{
    Customer johnCustomer = new Customer() {
        FirstName = "Mikhail",
        LastName = "Doe",
        Email = mikhail@Doesuccess.com
    };
    dbcontext.Customers.Add(johnCustomer);

    Customer annaCustomer = new Customer() {
        FirstName = "ivan",
        LastName = "Doe",
        Email = "mikhail@Doefail.com",
        Id = 123
    };
    dbcontext.Customers.Add(annaCustomer);
    dbcontext.SaveChanges();
}
```

Сначала создаем класс `Customer` и у него указываем имя, фамилию и email. Все хорошо, класс корректный и его можно создавать.

Потом создаем еще один класс `Customer` и указываем имя, фамилию, email и изменяем `id`. Это первичный ключ, и его просто так в базе данных изменять нельзя, поэтому такая операция, скорее всего, закончится ошибкой.

После добавления обоих клиентов в контекст происходит сохранение. Убедитесь, что журналирование включено, и запустите приложение, в результате чего вы должны увидеть следующее:

```

info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
    Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
    provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
    Executed DbCommand (32ms) [Parameters=[@p0='?' (Size = 4000),
    @p1='?' (Size = 4000), @p2='?' (Size = 4000), @p3='?' (Size = 4000)],
    CommandType='Text', CommandTimeout='30']
    SET NOCOUNT ON;
    INSERT INTO [Customers] ([Email], [FirstName], [LastName], [Phone])
    VALUES (@p0, @p1, @p2, @p3);
    SELECT [Id]
    FROM [Customers]
    WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
fail: Microsoft.EntityFrameworkCore.Database.Command[20102]
    Failed executing DbCommand (13ms) [Parameters=[@p0='?'
    (DbType = Int32), @p1='?' (Size = 4000), @p2='?' (Size = 4000),
    @p3='?' (Size = 4000), @p4='?' (Size = 4000)], CommandType='Text',
    CommandTimeout='30']
    INSERT INTO [Customers] ([Id], [Email], [FirstName], [LastName],
    [Phone])
    VALUES (@p0, @p1, @p2, @p3, @p4);
fail: Microsoft.EntityFrameworkCore.Update[10000]
    An exception occurred in the database while saving changes
    for context type 'CodeFirst.Data.MyStoreContext'.
    Microsoft.EntityFrameworkCore.DbUpdateException: An error occurred
    while saving the entity changes. See the inner exception for details.
    ---> Microsoft.Data.SqlClient.SqlException (0x80131904):
    Cannot insert explicit value for identity column in table
    'Customers' when IDENTITY_INSERT is set to OFF.
  
```

Информации очень много, но попробуем рассмотреть три основные операции, которые здесь выполнялись:

- ❑ первая операция **info** — это инициализация, которая прошла очень легко;
- ❑ вторая операция **info** — это попытка добавления первого пользователя, где корректные данные;
- ❑ третья операция **fail** — как я и ожидал, произошла ошибка и сохранения данных не произошло. Далее тут идет много информации, но это подробная информация об ошибке, а самое важное я в выводе выделил — это:

```

Cannot insert explicit value for identity column in table 'Customers'
when IDENTITY_INSERT is set to OFF (Здесь написано примерно следующее:
Нельзя вставлять явное значение для тождественной колонки в таблице
Клиентов, если IDENTITY_INSERT выключена.)
  
```

Отлично, все, как и ожидалось: одна строка добавлена, а другая сломалась. А действительно ли именно так произошло? Хотя **fail** светится только перед последней

операцией, реально база данных не сохранит ничего. Можете проверить — подключиться к базе данных и выполнить запрос:

```
select * from [Customers]
```

Та строка, которая должна была быть создана, реально отсутствует. А все дело в том, что по умолчанию метод `SaveChanges` сохраняет все или ничего, а это достигается выполнением операции в *транзакции*. Мы в журнале этого не видим, но в реальности будут выполнены четыре команды:

```
BEGIN TRANSACTION
INSERT INTO Customer ....
INSERT INTO Customer ....
COMMIT TRANSACTION
```

При таком подходе, если хотя бы одна операция вставки завершается ошибкой, то база откатывает все изменения.

Когда вы добавляете в контекст изменения, то обязательно думайте о том, что существуют транзакции, и сохраняйте данные тогда, когда считаете, что логически сохранение должно быть завершено. Сохранять двух клиентов в одной транзакции особого смысла нет — возможно, можно было сохранять сразу после добавления каждого из них в контекст.

Помимо транзакций, при сохранении данных EF может объединять несколько команд в одну. Если мы сохраняем в базу данных две записи, то проще и быстрее выполнить две SQL-команды `INSERT`, но если попытаться добавить в базу данных четыре и более записи, то тут есть способ оптимизации, и в одну SQL-команду добавить сразу все.

Давайте посмотрим, что будет, если в цикле создать четыре записи клиента и добавить их все сразу в один заход:

```
for (int i = 0; i < 4; i++)
{
    dbcontext.Customers.Add(new Customer()
    {
        FirstName = "User" + i.ToString(),
        LastName = "Doe",
        Email = "user" + i.ToString() + "@Doesuccess.com"
    });
}
dbcontext.SaveChanges();
```

Мало того что это будет выполняться в транзакции, так еще и SQL-команда получится достаточно интересная — специально созданная, чтобы добиться максимальной скорости. Запустите такой пример и посмотрите на результат — у меня в консоли появилась вот такая команда добавления записей в базу данных:

```
Executed DbCommand (58ms) [Parameters=[@p0='?' (Size = 4000), @p1='?' (Size = 4000), @p2='?' (Size = 4000), @p3='?' (Size = 4000), @p4='?' (Size = 4000), @p5='?' (Size = 4000), @p6='?' (Size = 4000), @p7='?'
```

```
(Size = 4000), @p8='?' (Size = 4000), @p9='?' (Size = 4000), @p10='?'
(Size = 4000), @p11='?' (Size = 4000), @p12='?' (Size = 4000), @p13='?'
(Size = 4000), @p14='?' (Size = 4000), @p15='?' (Size = 4000)],
CommandType='Text', CommandTimeout='30']

SET NOCOUNT ON;

DECLARE @inserted0 TABLE ([Id] int, [_Position] [int]);
MERGE [Customers] USING (
VALUES (@p0, @p1, @p2, @p3, 0),
(@p4, @p5, @p6, @p7, 1),
(@p8, @p9, @p10, @p11, 2),
(@p12, @p13, @p14, @p15, 3)) AS i ([Email], [FirstName], [LastName],
[Phone], _Position) ON 1=0

WHEN NOT MATCHED THEN
INSERT ([Email], [FirstName], [LastName], [Phone])
VALUES (i.[Email], i.[FirstName], i.[LastName], i.[Phone])
OUTPUT INSERTED.[Id], i._Position
INTO @inserted0;

SELECT [i].[Id] FROM @inserted0 i
ORDER BY [i].[_Position];
```

Здесь создается табличная переменная. В эту табличную переменную помещаются через параметры данные. Потом выполняется запрос SQL на вставку данных и, наконец, выборка созданных для новых записей уникальных идентификаторов. На первый взгляд, все выглядит сложно, но в реальности выполняются только три команды — несмотря на то, сколько строк одновременно будут вставляться.

Таким образом EF автоматически реализовывает оптимизацию вставки большого количества данных, и это плюс. Я люблю контроль и чистый SQL, но отдаю уважение EF за такую оптимизацию, потому что подобный запрос писать вручную может быть не так уж и просто новичкам.

Количество записей, которые могут объединяться в одну транзакцию, не бесконечно. Если вам нужно будет вставить в базу данных 1000 строк, то это не значит, что они будут вставляться все сразу, потому что есть пределы на то, сколько параметров будет работать эффективно. Где-то в документации я видел, что максимальное количество параметров равно 2100, а максимальное количество параметров по умолчанию — 42, так что вот количество строк на один запрос вы настроить сможете. А разбиение 1000 строк на блоки по 42 будет реализовываться автоматически — об этом нам заботиться не нужно.

15.14. Погружение в запросы

Очень сложно выстраивать объяснение работы с EF, не прыгая по темам. Надо было объяснить, как можно добавить данные в базу, чтобы показать хотя бы простые запросы к данным и их отображение. Но прежде чем показать более сложные запросы, нужно было поговорить о связях и рассказать, как добавлять данные за одну транзакцию.

Теперь стоит вернуться назад и снова обратиться к вопросам доступа к данным и их отображения. В разделах 15.4 и 15.7 я уже немного говорил об этом, а теперь настало время рассмотреть их более основательно.

В случае с EF для доступа к данным в базе можно использовать LINQ-запросы, что дает как преимущество, так и недостаток. Преимущество — вы получаете гибкость декларативного подхода. Недостаток — вы теряете контроль над тем, какой SQL-запрос в реальности будет выполнен относительно базы данных.

Взглянем на следующий фрагмент кода:

```
using (var dbcontext = new MyStoreContext())
{
    Console.WriteLine($"Имя \t Фамилия");
    foreach (Customer c in dbcontext.Customers)
    {
        Console.WriteLine($"{c.FirstName} \t {c.LastName}");
    }
}
```

Когда здесь будет выполнен запрос `SELECT` к базе данных? На этот вопрос ответить достаточно просто — когда мы начинаем выполнять цикл, будет открыто соединение с базой данных и тут же выполнен запрос.

Немного усложним задачу:

```
using (var dbcontext = new MyStoreContext())
{
    var customers = dbcontext.Customers;

    Console.WriteLine($"Имя \t Фамилия");
    foreach (Customer c in customers)
    {
        Console.WriteLine($"{c.FirstName} \t {c.LastName}");
    }
}
```

Здесь сначала создается переменная, в которую попадает свойство `Customers`, а уже затем мы выводим название колонок и начинаем перебирать клиентов.

Когда теперь выполнится запрос? На самом деле ничего не изменится. Если посмотреть на консоль при включенном журналировании, то мы увидим:

```
Имя    Фамилия
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options: None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (17ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']
      SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
      [c].[Phone]
```

```
FROM [Customers] AS [c]
John      Doe
Anna      Doe
User0     Doe
```

Сначала отобразились имена колонок, а потом уже произошла инициализация EF Core, — т. е. запрос выполнится при первом обращении в цикле.

Еще усложняем задачу и добавляем LINQ:

```
using (var dbcontext = new MyStoreContext())
{
    var customers = dbcontext.Customers;
    var result = customers.Where(m => m.LastName == "Doe");
    result = result.Where(m => m.FirstName == "Anna");

    Console.WriteLine($"Имя \t Фамилия");
    foreach (Customer c in result)
    {
        Console.WriteLine($"{c.FirstName} \t {c.LastName}");
    }
}
```

В консоли вы должны увидеть примерно следующее:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5'
                                         with options: None
Имя      Фамилия
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (18ms) [Parameters=[], CommandType='Text',
      CommandTimeout='30']
      SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
      [c].[Phone]
      FROM [Customers] AS [c]
      WHERE ([c].[LastName] = N'Doe') AND ([c].[FirstName] = N'Anna')
Anna      Doe
```

Сначала произошла инициализация EF Core, потом появились имена колонок, а потом уже выполнен запрос. Хотя в коде у нас сначала идут два LINQ-запроса, а только потом предусмотрено отображение имен колонок, в реальности запрос выполнен только тогда, когда началось выполнение цикла, и только один раз. Единственное, что поменялось, — инициализация произошла сразу же при первом обращении к LINQ.

Тут очень важно подчеркнуть: хотя у меня два LINQ-запроса с фильтрацией сначала по фамилии, а потом по имени, был выполнен только один запрос к базе данных, который фильтровал сразу и по фамилии, и по имени:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE ([c].[LastName] = N'Doe') AND ([c].[FirstName] = N'Anna')
```

Причем в запросе значения четко прописаны прямо в SQL и не использовались никакие параметры. Это плохо с точки зрения производительности, потому что база данных не будет кешировать план выполнения. Вместо жестко прописанных значений лучше использовать параметры.

Самый простой способ решить эту проблему — завести C#-переменные и использовать их в LINQ-запросах:

```
using (var dbcontext = new MyStoreContext())
{
    var customers = dbcontext.Customers;

    string lastname = "Doe";
    string firstname = "Anna";

    var result = customers.Where(m => m.LastName == lastname);
    result = result.Where(m => m.FirstName == firstname);

    Console.WriteLine($"Имя \t Фамилия");
    foreach (Customer c in result)
    {
        Console.WriteLine($"{c.FirstName} \t {c.LastName}");
    }
}
```

В этот раз EF выполнит запрос с параметрами @__lastname_0 и @__firstname_1:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName],
       [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE ([c].[LastName] = @__lastname_0)
      AND ([c].[FirstName] = @__firstname_1)
```

При создании SQL-переменных EF берет имя C#-переменной в нашем коде и последовательно добавляет в конец индекс, начиная с нуля, чтобы обезопасить от повтора.

А что если в коде программы выполняются два цикла `foreach`, которые перебирают данные? Просто скопируйте цикл в коде дважды, и в консоли мы увидим, что EF Core реально дважды направил запрос к базе данных:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5'
                                         with options: None
```

Имя Фамилия

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (32ms) [Parameters=[@__lastname_0='?'
      (Size = 4000), @__firstname_1='?' (Size = 4000)], CommandType='Text',
      CommandTimeout='30']
```

```
SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
                                             [c].[Phone]
FROM [Customers] AS [c]
WHERE ([c].[LastName] = @__lastname_0) AND ([c].[FirstName] =
                                             @__firstname_1)

Anna      Doe
Имя       Фамилия
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (3ms) [Parameters=[@__lastname_0='?' (Size = 4000),
      @__firstname_1='?' (Size = 4000)], CommandType='Text',
      CommandTimeout='30']
SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
                                             [c].[Phone]
FROM [Customers] AS [c]
WHERE ([c].[LastName] = @__lastname_0) AND ([c].[FirstName] =
                                             @__firstname_1)

Anna      Doe
```

Инициализация произошла только один раз, а вот обращений к базе данных было два, потому что у нас два цикла.

Иногда возникает необходимость работать с одним и тем же набором данных дважды: в первый раз мы можем захотеть просмотреть данные, а во второй — подсчитать что-то или сделать какие-то манипуляции. Если запрос простой и быстрый, как в показанном случае, тогда то, что мы выполняем его дважды, может и не быть проблемой. А если запрос очень сложный и база данных тратит на его выполнение десятки секунд? Да, даже пара секунд уже станут проблемой, потому что когда дважды выполняется запрос длительностью в 2 секунды, то страница для веб-приложения будет загружаться минимум 4 секунды. Вы любите, когда страницы сайта загружаются 4 и более секунд?

Даже если запрос выполняется очень быстро, отправка двух отдельных SQL-запросов может вернуть разные данные, потому что они могли уже быть изменены за это время на сервере кем-то другим. Это тоже может привести к проблемам.

Так что нужно как-то сказать EF Core, в какой момент выполнить запрос и запомнить результат.

Наведите мышь на переменную `result` и посмотрите на ее тип — вы должны увидеть `IQueryable<Customer>`.

`IQueryable` и `IEnumerable` — это интерфейсы, которые не хранят результат, а просто предоставляют доступ к перечислению данных. Каждый раз, когда цикл начинает перечисление, интерфейсы приводят к тому, что EF запрашивает свежие данные с сервера и перечисляет их.

Чтобы изменить этот подход, нужно превратить данные в список с помощью `ToList`. Можно использовать и любой другой тип данных, который их удерживает, — например, `ToArray`, но чаще просто берут `ToList`, потому что он позволяет потом добавлять и удалять данные. Давайте подправим наш код:

```
var resultlist = result.Where(m => m.FirstName == firstname).ToList();

Console.WriteLine($"Имя \t Фамилия");
foreach (Customer c in resultlist)
    Console.WriteLine($"{c.FirstName} \t {c.LastName}");

Console.WriteLine($"Имя \t Фамилия");
foreach (Customer c in resultlist)
    Console.WriteLine($"{c.FirstName} \t {c.LastName}");
```

В результате в консоли мы увидим:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options:
      None
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (33ms) [Parameters=[@__lastname_0='?'
      (Size = 4000), @__firstname_1='?' (Size = 4000)], CommandType='Text',
      CommandTimeout='30']
      SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
      [c].[Phone]
      FROM [Customers] AS [c]
      WHERE ([c].[LastName] = @__lastname_0) AND ([c].[FirstName] =
      @__firstname_1)
```

Имя	Фамилия
Anna	Doe

```
Имя      Фамилия
Anna     Doe
```

Запрос явно выполнялся только один раз, а вот отображение произошло дважды именно потому, что `List` хранит данные в памяти.

Когда мы вызываем метод `ToList` или `ToArray`, говорят, что мы *материализуем запрос*. Это значит, что LINQ-запрос превращается в SQL-запрос — он выполняется в отношении базы данных, и данные возвращаются приложению и сохраняются в его памяти.

Бывают случаи, когда SQL Server генерирует запрос, который не очень эффективно выполняется на сервере базы данных. Или, может, нам нужно получить из базы данных всех клиентов по фамилии `Doe`, а потом фильтровать эти данные разным способом. Как поступить?

Можно материализовать запрос и потом фильтровать данные уже в памяти компьютера:

```
var customers = dbcontext.Customers;
string lastname = "Doe";
var result = customers.Where(m => m.LastName == lastname).ToList();

Console.WriteLine($"Имя \t Фамилия");
foreach (Customer c in result.Where(m => m.FirstName == "Anna"))
    Console.WriteLine($"{c.FirstName} \t {c.LastName}");
```



```
Console.WriteLine($"Имя \t Фамилия");  
foreach (Customer c in result.Where(m => m.FirstName == "John"))  
    Console.WriteLine($"{{c.FirstName}} \t {{c.LastName}}");
```

В этом коде на третьей строке запрос материализуется и данные попадают в переменную `result`. Так как мы материализовали данные в список, теперь `result` будет не `IQueryable`, а `List<Customer>`.

Выполните код и посмотрите на результат в терминале:

```
info: Microsoft.EntityFrameworkCore.Infrastructure[10403]  
      Entity Framework Core 6.0.6 initialized 'MyStoreContext' using  
      provider 'Microsoft.EntityFrameworkCore.SqlServer:6.0.5' with options:  
      None  
  
info: Microsoft.EntityFrameworkCore.Database.Command[20101]  
      Executed DbCommand (30ms) [Parameters=[@__lastname_0='?'  
      (Size = 4000)], CommandType='Text', CommandTimeout='30']  
      SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],  
      [c].[Phone]  
      FROM [Customers] AS [c]  
      WHERE [c].[LastName] = @__lastname_0  
  
Имя      Фамилия  
Anna      Doe  
Имя      Фамилия  
John      Doe
```

Запрос выполнился только один раз в тот момент, когда мы выполняем материализацию:

```
var result = customers.Where(m => m.LastName == lastname).ToList();
```

В этот момент мы фильтруем только по фамилии, поэтому SQL-запрос тоже фильтрует только по фамилии.

Когда же выполняется цикл, то я фильтрую данные еще раз с помощью LINQ:

```
foreach (Customer c in result.Where(m => m.FirstName == "Anna"))
```

Но в этот раз запроса к базе данных нет. Дело в том, что мы выполняем LINQ относительно `result`, который является списком, а не `IQueryable`, поэтому здесь фильтрация будет происходить уже в памяти компьютера.

Когда материализовывать? Все зависит от ситуации. Лучше писать LINQ-запросы так, чтобы они возвращали только необходимые данные, и чем меньше, тем лучше. Если запрос возвращает много данных, то он может дольше выполняться на сервере, и больше будет копироваться данных между базой данных и приложением. Поэтому лучше фильтровать данные с помощью SQL, а не в приложении. Но бывают случаи, когда все же эффективнее сделать какие-то агрегации в приложении, а не в базе данных. Тут все определяется опытом.

Еще одна разница между перебором данных в цикле и использованием `ToList` — во времени, в течение которого будет открыто соединение, и используемой памяти. Когда идет перебор в цикле, то открывается соединение, и мы читаем данные по-

строчно. Пока мы читаем, соединение с базой данных остается открытым, при этом открыт и курсор чтения. Это приводит к двум проблемам:

- ❑ если в цикле выполняются долгие операции, то соединение с базой может быть открыто продолжительное время, а с точки зрения производительности лучше долго не держать его открытым;
- ❑ пока вы читаете одни данные и в цикле попытаетесь обратиться к другим данным, то может возникнуть ошибка множественного курсора. Решается это изменением конфигурации в файле `web.config`, но лучше все же не использовать множественные курсоры.

При использовании `ToList` или `ToArray` соединение будет открыто только на время копирования данных в память списка/массива. Но может потребоваться больше памяти для хранения всех данных.

Так что если цикл работает очень долго, то перед его началом, может быть, стоит вызвать `ToList`, чтобы материализовать запрос: скопировать данные в память и закрыть соединение.

Мы уже не раз использовали LINQ-запросы, чтобы обращаться к данным. В случае с EF вы можете использовать все мощности LINQ для фильтрации данных. Если при написании LINQ-запроса использовать строку в чистом виде:

```
var result = customers.Where(m => m.LastName == "Doe").ToList();
```

то и в SQL она попадет в чистом виде:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE [c].[LastName] = N'Doe')
```

Но это неэффективно с точки зрения производительности, потому что SQL-сервер не может кешировать план выполнения таких запросов, и лучше использовать параметризированные запросы.

Чтобы фамилия превратилась в параметр в запросе, нужно просто в C#-коде тоже использовать переменную — просто `C#`:

```
string lastname = "Doe";
var result = customers.Where(m => m.LastName == lastname).ToList();
```

Вот теперь EF сгенерирует параметризированный запрос:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName],
       [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE [c].[LastName] = @__lastname_0
```

Тут же в журнале можно также увидеть и описание запроса:

```
Executed DbCommand (53ms)
[
  Parameters= [
    @__lastname_0='?'
```

```
(Size = 4000)
],
CommandType='Text',
CommandTimeout='30'
]
```

Я немного переформатировал текст, чтобы можно было увидеть четыре момента, о которых нам хотят сообщить. В первой строке нам говорят, что запрос выполнялся 53 миллисекунды и что был выполнен реально объект `DbCommand` — это класс низкого уровня в ADO.NET для выполнения запросов к базе данных. То есть EF сгенерировал SQL-запрос и для его выполнения воспользовался объектом `DbCommand`.

Потом идут установленные свойства. Первое из них — `Parameters`, и тут видно имя параметра: `@__lastname_0` и его размер: 4000 символов. А второе — `CommandType` в `Text`. Это свойство также может принимать значения: `StoredProcedure` — для выполнения хранимых процедур и `TableDirect` — когда просто указывается имя таблицы.

На выполнение запроса было выделено 30 секунд (об этом говорит свойство `CommandTimeout`).

После каждого параметра C# будет добавлять индекс, чтобы избежать конфликтов. А как могут возникнуть эти конфликты? — ведь мы не можем в одном методе создать две переменные с одним и тем же именем! В одном методе не можем, зато в двух разных можем. Давайте создадим метод, который будет возвращать `IQueryable` и фильтровать по фамилии:

```
IQueryable<Customer> GetQuery(MyStoreContext dbcontext)
{
    string lastname = "Doe";
    return dbcontext.Customers.Where(m => m.LastName == lastname);
}
```

Метод фильтрует по параметру `lastname`, и если мы сделаем то же самое в другом месте, то получится, что мы фильтруем дважды по одному и тому же параметру:

```
string lastname = "Doe";
var result = GetQuery(dbcontext).Where(m => m.LastName == lastname);
```

Но за счет того, что сервер добавляет индекс параметра, мы в безопасности:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE ([c].[LastName] = @__lastname_0) AND ([c].[LastName] = @__lastname_1)
```

потому что были созданы два разных параметра.

Возможность строить основу для запроса в одном методе и потом использовать эту основу для дальнейшей фильтрации в другой — это огромный плюс EF Core. Да, то же самое можно сделать и с чистым SQL, но тогда не будет проверки на ошибки запроса SQL. А вот EF генерирует запрос перед самым выполнением и всегда генерирует его правильно.

Следующий важный момент, который надо отметить по поводу выполнения запросов EF, — когда происходит фиксация того, что будет выполнено относительно базы данных, а что будет выполнено на стороне сервера приложений. Мы уже знаем, что `IQueryable` и `IEnumerable` — это интерфейсы, которые не хранят результата, а просто предоставляют доступ к перечислению данных. Как уже отмечалось ранее, каждый раз, когда цикл начинает перечисление, интерфейсы приводят к тому, что EF запрашивает свежие данные с сервера и перечисляет их.

Но в чем между ними разница? И зачем нужны два интерфейса? В предыдущем примере я создал метод, который возвращает `IQueryable`, и это не случайно. Дело в том, что этот интерфейс возвращает запрос (`Query`), который можно изменять, что мы и делаем.

А что, если поменять `IQueryable` на `IEnumerable`:

```
IEnumerable<Customer> GetQuery(MyStoreContext dbContext)
{
    string lastname = "Doe";
    return dbContext.Customers.Where(m => m.LastName == lastname);
}
```

Теперь этот метод возвращает `IEnumerable`, а это данные, а не запрос. Пусть они не хранятся в памяти и каждый раз запрашиваются с сервера, но это все же данные. То есть метод возвращает данные `IEnumerable`, и теперь в следующей строке мы фильтруем данные, а не запрос:

```
var result = GetQuery(dbContext).Where(m => m.LastName == lastname);
```

Выполните этот код и в терминале вы увидите:

```
SELECT [c].[Id], [c].[Email], [c].[FirstName],
       [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE [c].[LastName] = @__lastname_0
```

Теперь к базе данных был направлен запрос только с одним параметром, он вернул данные, и после этого они были отфильтрованы с помощью LINQ уже на клиенте.

Еще раз напоминаю, что оба эти интерфейса не хранят данные, а предоставляют интерфейс доступа к данным. Если мы будем дважды обращаться к данным, то будет дважды выполняться и запрос к базе данных.

`IEnumerable` стоит использовать тогда, когда мы хотим сказать, что пора уже зафиксировать SQL-запрос и далее работать с данными на стороне сервера приложений. То есть если вы хотите модифицировать запрос, то нужно использовать `IQueryable`. Если же вы не планируете модифицировать запрос в других методах, то лучше возвращать данные `IEnumerable`.

15.15. Асинхронный доступ

При обращении к данным лучше всего использовать не блокирующие режимы `ToListAsync()` или `ToArrayAsync()`. В этом случае .NET сможет более эффективно использовать потоки:

```
var result = await GetQuery(dbcontext)
    .Where(m => m.LastName == lastname).ToListAsync();
```

или

```
var result = await dbcontext.Customers
    .Where(m => m.LastName == lastname).ToListAsync();
```

Если метод `ToToListAsync` окажется недоступен, нужно подключить следующее пространство имен:

```
using Microsoft.EntityFrameworkCore;
```

Мы уже использовали функцию `FirstOrDefault`, которая возвращает результат из базы данных в виде одной записи или, если ничего не найдено, то `null`. Для этого способа также есть асинхронная версия:

```
var result = await dbcontext.Customers
    .Where(m => m.LastName == lastname)
    .FirstOrDefaultAsync();
```

В результате будет сформирован запрос:

```
SELECT TOP(1) [c].[Id], [c].[Email], [c].[FirstName],
    [c].[LastName], [c].[Phone]
FROM [Customers] AS [c]
WHERE [c].[LastName] = @__lastname_0
```

Здесь такой же запрос `SELECT`, как и в случае с возвращением списка записей, но разница в том, что после `SELECT` идет `TOP(1)`, который ограничивает вывод только одной записью. В коде ожидается только одна запись, поэтому в запросе давать шанс вернуть несколько записей смысла нет.

Рекомендую всегда применять асинхронный подход — особенно для веб-приложений. Я же в дальнейшем изложении буду упрощать описания и использовать в основном `ToList`.

15.16. Поиск по шаблону

Проверка на равенство — это просто, а как сделать проверку типа `Like`? В `C#` есть возможность проверять строку на содержание подстроки с помощью `Contains`, поэтому его и используем:

```
var result = GetQuery(dbcontext).Where(m => m.LastName.Contains("o"));
```

А что если нам нужно найти все фамилии, которые начинаются на букву `D`? Можно попытаться играть с методами `C#`, но проще применить специальные функции.

Подключите пространство имен `EntityFrameworkCore`:

```
using Microsoft.EntityFrameworkCore;
```

и можно будет воспользоваться функцией `EF.Functions.Like`:

```
var result = GetQuery(dbcontext).Where(m =>
    EF.Functions.Like(m.LastName, "D%"));
```

15.17. Удаление данных

Мы уже достаточно хорошо рассмотрели поиск и поговорили про обновление данных, и теперь пора познакомиться с еще одной важной операцией — удалением.

Для поиска данных мы используем `dbcontext.Customers`, который является набором данных, и вполне логично было бы предположить, что удаление из базы данных будет простым удалением записи из этого набора. На самом деле так оно и есть.

В следующем примере я нахожу первую попавшуюся запись с фамилией `Doe` и удаляю ее:

```
using (var dbcontext = new MyStoreContext())
{
    string lastname = "Doe";
    var customer = await dbcontext.Customers
        .Where(m => m.LastName == lastname)
        .FirstOrDefaultAsync();

    Console.WriteLine("Клиентов = " + dbcontext.Customers.Count());

    if (customer != null)
    {
        dbcontext.Customers.Remove(customer);
        dbcontext.SaveChanges();
    }

    Console.WriteLine("Клиентов = " + dbcontext.Customers.Count());
}
```

Если исключить поиск, то удаление происходит в две строки: удалить запись из набора и сохранить изменения в базе данных:

```
dbcontext.Customers.Remove(customer);
dbcontext.SaveChanges();
```

Чтобы стало видно, что данные удалены, я вывожу количество клиентов в базе данных до удаления и после него. У меня до удаления было 6 клиентов, а после удаления их стало пять.

Количество записей в базе данных можно узнать, просто обратившись к методу `Count()`:

```
dbcontext.Customers.Count()
```

Посмотрев на консоль, мы увидим простой SQL-запрос на удаление `DELETE FROM table`, после которого возвращается количество записей, которые были удалены:

```
info: Microsoft.EntityFrameworkCore.Database.Command[20101]
      Executed DbCommand (11ms) [Parameters=[@p0='?' (DbType = Int32)],
      CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
DELETE FROM [Customers]
```

```
WHERE [Id] = @p0;  
SELECT @@ROWCOUNT;
```

15.18. Отслеживание изменений

Мы уже увидели, что изменение данных происходит достаточно просто: получаем данные из базы, изменяем свойства полученных объектов и вызываем `SaveChanges()` контекста базы данных. Контекст отслеживает любые изменения и автоматически генерирует запросы для изменения только измененных колонок. Например:

```
var customer = await dbcontext.Customers  
    .Where(m => m.LastName == "Doe")  
    .FirstOrDefaultAsync();  
  
if (customer != null)  
{  
    customer.FirstName = "Updated";  
    dbcontext.SaveChanges();  
}
```

Здесь я нахожу первого попавшегося клиента с фамилией `Doe` и изменяю его имя на `Updated`. Если выполнить этот код, то EF выполнит следующий запрос относительно базы данных:

```
SET NOCOUNT ON;  
UPDATE [Customers] SET [FirstName] = @p0  
WHERE [Id] = @p1;  
SELECT @@ROWCOUNT;
```

Это хорошо, но только требует ресурсов. Контексту необходимо отслеживать не только факт изменения объекта, но и какие именно были сделаны изменения, чтобы впоследствии сгенерировать запрос, который будет обновлять в базе данных только измененные данные.

Прежде чем мы продолжим разговор, обновим имена всех клиентов, чтобы они имели формат `User X`, для чего выполним следующий код:

```
var customers = dbcontext.Customers.ToList();  
for (int i = 0; i < customers.Count(); i++)  
    customers[i].FirstName = "User " + i.ToString();  
dbcontext.SaveChanges();
```

Но если вы получаете данные и не планируете их изменять, то можно отключить отслеживание. Для этого при выполнении запроса можно добавить `.AsNoTracking()`:

```
var customer = await dbcontext.Customers  
    .Where(m => m.LastName == "Doe")  
    .AsNoTracking()  
    .FirstOrDefaultAsync();
```

```

if (customer != null)
{
    customer.FirstName = "Updated";
    dbcontext.SaveChanges();
}

var customers = dbcontext.Customers.ToList();
for (int i = 0; i < customers.Count(); i++)
    Console.WriteLine("Имя = " + customers[i].FirstName);

```

Здесь я снова ищу в базе данных клиента с фамилией Doe, но при этом отключаю трекинг, добавляя `AsNoTracking`. Затем пробую изменить имя, если хотя бы одна запись найдена. Ну и в конце вывожу имена всех клиентов, чтобы мы могли увидеть — произошли изменения или нет. Изменений нет — нет ни одного клиента с именем Updated.

Так что если вы не планируете изменять данные, то я рекомендую добавлять `AsNoTracking` для повышения производительности, чтобы EF не создавал в фоне ничего дополнительного для отслеживания изменений.

Если вы работаете над API, который в основном читает данные и очень редко изменяет их, то можно изменить поведение контекста по умолчанию так, чтобы он изначально не отслеживал изменения. А лучше создать два контекста: один для изменения данных, а другой — только для чтения:

```

internal class MyStoreContextReadOnly : MyStoreContext
{
    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);
        optionsBuilder
            .UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);
    }
}

```

Здесь я создаю новый контекст `MyStoreContextReadOnly`, который наследует существующий `MyStoreContext` и переопределяет метод `OnConfiguring`, где сначала вызывается такой же метод предка (там у нас создается строка подключения) и изменяется поведение отслеживания по умолчанию на `QueryTrackingBehavior.NoTracking`.

Теперь посмотрим на следующий код:

```

using (var dbcontext = new MyStoreContext())
{
    var customers = dbcontext.Customers.ToList();
    for (int i = 0; i < customers.Count(); i++)
        customers[i].FirstName = "User " + i.ToString();
    dbcontext.SaveChanges();
}

```



```
using (var dbcontext = new MyStoreContextReadOnly()) {
    var customer = await dbcontext.Customers
        .Where(m => m.LastName == "Doe")
        .FirstOrDefaultAsync();

    if (customer != null)
    {
        customer.FirstName = "Updated";
        dbcontext.SaveChanges();
    }

    var customers = dbcontext.Customers.ToList();
    for (int i = 0; i < customers.Count(); i++)
        Console.WriteLine("Имя = " + customers[i].FirstName);
}
```

Здесь сначала создается контекст, который поддерживает отслеживание изменений, и все имена клиентов генерируются в соответствии с форматом. Эта операция пройдет успешно, и изменения будут сохранены в базе данных.

Потом создается контекст только для чтения, и в нем попытка изменить имя уже будет безуспешной — имя не изменится, потому что у этого контекста отслеживание отключено.

Мне кажется, что два контекста: для записи и для чтения — будут удобным вариантом в большинстве случаев.

Есть еще один способ — сразу после создания контекста установить его свойство `ChangeTracker.QueryTrackingBehavior`:

```
using (var dbcontext = new MyStoreContextReadOnly()) {
    dbcontext.ChangeTracker.QueryTrackingBehavior =
        QueryTrackingBehavior.TrackAll;
    var customer = await dbcontext.Customers
        .Where(m => m.LastName == "Doe")
        .FirstOrDefaultAsync();
    . . .
    . . .
}
```

Здесь я создаю контекст только для чтения, но сразу после создания меняю поведение отслеживания на `QueryTrackingBehavior.TrackAll` и таким образом включаю отслеживание. В этом примере изменения будут сохранены в базе данных.

Это действует и в противоположном направлении: можно создать контекст, который по умолчанию отслеживает изменения, и изменить его поведение на `QueryTrackingBehavior.NoTracking`:

```
Dbcontext.ChangeTracker.QueryTrackingBehavior =
    QueryTrackingBehavior.NoTracking;
```

Изменять этот параметр нужно до того, как выполнится запрос к базе данных. Если вы измените поведение `ChangeTracker` после того, как получены данные, и перед началом их изменения, то это не сработает.

Таким образом мы рассмотрели три способа изменения поведения отслеживания — вам остается только выбрать тот, который подходит в конкретном случае. Можно, конечно, выбрать один и всегда использовать только его, но я бы рекомендовал выбирать вариант применительно к каждой конкретной ситуации.

Еще один вопрос производительности. В приведенном ранее коде я изменил только определенные свойства, и EF сгенерировал запрос на изменения только их:

```
SET NOCOUNT ON;
UPDATE [Customers] SET [FirstName] = @p0
WHERE [Id] = @p1;
SELECT @@ROWCOUNT;
```

У контекста EF есть метод `Update`, который тоже позволяет обновлять данные даже в том случае, когда нет трекинга:

```
using (var dbcontext = new MyStoreContext())
{
    var customers = dbcontext.Customers.AsNoTracking().ToList();
    for (int i = 0; i < customers.Count(); i++)
    {
        customers[i].FirstName = "User " + i.ToString();
        dbcontext.Update(customers[i]);
    }
    dbcontext.SaveChanges();
}
```

Здесь сначала я отключаю трекинг, когда обращаюсь к данным `dbcontext.Customers.AsNoTracking().ToList()`. Но я все еще могу обновить данные с помощью метода `Update` — просто изменить имя и вызвать метод `Update`:

```
dbcontext.Update(customers[i]);
```

В этом случае, несмотря на то, что отслеживание отключено, данные будут обновлены. Но лучше так не делать! Включите журналирование и посмотрите, какой запрос сгенерировал EF:

```
SET NOCOUNT ON;
UPDATE [Customers] SET [Email] = @p0, [FirstName] = @p1, [LastName] = @p2,
                                                             [Phone] = @p3
WHERE [Id] = @p4;
SELECT @@ROWCOUNT;
```

Несмотря на то что я изменил только имя, SQL обновляет все колонки. И проблема не в том, что я отключил отслеживание, а в том, что метод `Update` игнорирует отслеживание. Даже если включить отслеживание и убрать `AsNoTracking()`, EF все равно сгенерирует запрос на обновление всех колонок, потому что так ведет себя `Update`.

Вызывайте `Update` только в тех случаях, когда вы действительно нуждаетесь в обновлении всех колонок и обновлении в нужное состояние даже тех колонок, которые вы не меняли. База данных может быть умной и физически не обновлять в своем хранилище данные, но ей все равно придется проверить — нужно ли обновлять или нет.

15.19. Погружение в связи

Мы уже неплохо окунулись в описание и создание связанных таблиц. А после этого — в вопрос добавления и получения данных из базы.

Я уже говорил о том, что для получения связанных данных нужно использовать метод `Include` и в нем указать, какие именно данные вы хотите получить. По умолчанию EF не загружает связанные данные, потому что это может привести к серьезным проблемам производительности.

Пусть в нашей базе данных имеется клиент, у которого есть ссылка на счета, и этих счетов может быть очень много. У каждого счета есть детали — позиции, которые были куплены. Каждая позиция ссылается на товар. И если мы хотим найти этого клиента, чтобы отобразить его имя и адрес, то с помощью LINQ выполняем следующий код:

```
Customer? johnCustomer = dbcontext.Customers
    .Where(m => m.Email == John@Doe.com)
    .FirstOrDefault();
```

Если EF начнет загружать все связанные данные, то ему придется загрузить все счета, детали счетов и данные о товарах. Но EF загружает только данные клиента, а если вам нужны связанные данные, то вы должны это явно указать с помощью `Include`. В следующем примере я говорю, что нужно включить данные о счетах:

```
Customer? johnCustomer = dbcontext.Customers
    .Where(m => m.Email == John@Doe.com)
    .Include(m => m.Orders)
    .FirstOrDefault();
```

Давайте выполним этот код и посмотрим, какой запрос сгенерирует EF:

```
SELECT [t].[Id], [t].[Email], [t].[FirstName], [t].[LastName], [t].[Phone],
       [o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified]
FROM (
    SELECT TOP(1) [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
                  [c].[Phone]
    FROM [Customers] AS [c]
    WHERE [c].[Email] = N'John@Doe.com'
) AS [t]
LEFT JOIN [Orders] AS [o] ON [t].[Id] = [o].[CustomerId]
ORDER BY [t].[Id]
```

На мой взгляд, вполне ожидаемо — здесь выполняется один SQL-запрос с `LEFT JOIN`. Именно левое объединение данных, а не `INNER`.

Но что если у клиента 100 счетов? База данных должна будет вернуть 100 строк счетов и в каждой из них указать имя клиента, фамилию, телефон, Email. Одна и та же информация в 100 строк приведет к тому, что с базы данных на сервер приложения будет перетекать очень много информации.

Давайте создадим нового клиента с двумя счетами:

```
if (!dbcontext.Customers.Any(m => m.Email == "with@orders.com"))
{
    dbcontext.Customers.Add(
        new Customer()
        {
            Email = "with@orders.com",
            FirstName = "With",
            LastName = "Orders",
            Phone = "123",
            Orders = new List<Order> {
                new Order() {
                    Created = DateTime.Now.AddMinutes(10),
                    Modified = DateTime.Now
                },
                new Order() {
                    Created = DateTime.Now.AddMinutes(25),
                    Modified = DateTime.Now
                },
            }
        });
    dbcontext.SaveChanges();
}
```

Если клиент с Email `with@orders.com` не найден, то нужно его добавить. И тут я использую более сокращенную версию добавления клиента и счетов как одного большого объекта. То есть при создании объекта клиента я тут же создал в нем список с двумя счетами.

Когда я рассматривал добавление записей в предыдущий раз, то там все происходило в два этапа: добавить клиента — сохранить, найти клиента (чтобы получить его ID), создать объект счета и назначить связь:

```
Customer? johnCustomer = dbcontext.Customers.Where(
    m => m.Email == "John@Doe.com").FirstOrDefault();
dbcontext.Orders.Add(
    new Order()
    {
        CustomerId = johnCustomer.Id,
        Created = DateTime.Now,
        Modified = DateTime.Now
    });
dbcontext.SaveChanges();
```

Просто чаще приходится добавлять связанные данные к существующим, поэтому я сначала показал такой способ. Сейчас я просто генерирую данные, и удобнее использовать один большой объект. Ну а как вы будете поступать — зависит от вашей ситуации.

Теперь выполним запрос, который сгенерировал EF, только поменяем Email на только что созданного пользователя (рис. 15.7):

```
SELECT [t].[Id], [t].[Email], [t].[FirstName], [t].[LastName], [t].[Phone],
[o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified]
FROM (
    SELECT TOP(1) [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
    [c].[Phone]
    FROM [Customers] AS [c]
    WHERE [c].[Email] = N'with@orders.com'
) AS [t]
LEFT JOIN [Orders] AS [o] ON [t].[Id] = [o].[CustomerId]
ORDER BY [t].[Id]
```

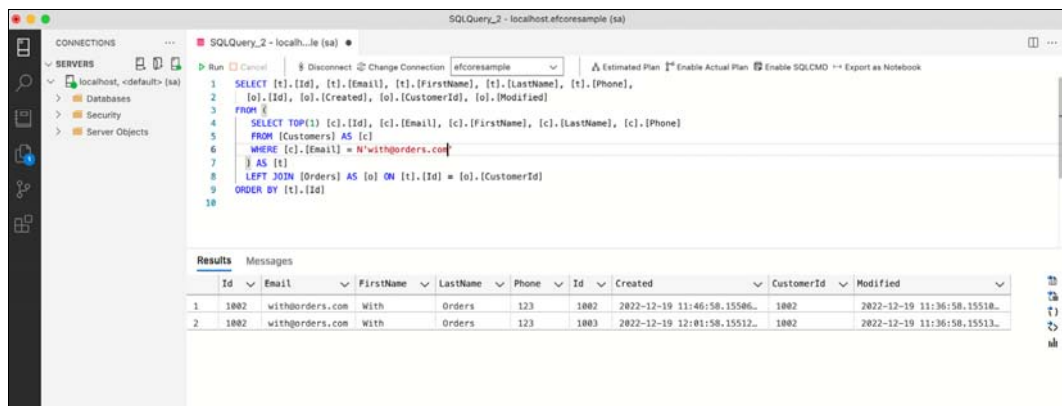


Рис. 15.7. Повторение данных

Я вижу здесь две строки, в которых первые пять колонок приходят из таблицы Customers, поэтому данные в них повторяются. Это может быть плохо с точки зрения производительности. В таких случаях лучше выполнить два запроса: получить одну строку из Customers и все счета из Orders. И это достаточно просто сделать, если добавить к нашему запросу .AsSplitQuery():

```
Customer? customer = dbcontext.Customers
    .Where(m => m.Email == "with@orders.com")
    .AsSplitQuery()
    .Include(m => m.Orders)
    .FirstOrDefault();
```

Выполняем этот код и видим два запроса. Первый ищет клиента:

```
SELECT TOP(1) [c].[Id], [c].[Email], [c].[FirstName], [c].[LastName],
[c].[Phone]
```

```
FROM [Customers] AS [c]
WHERE [c].[Email] = N'with@orders.com'
ORDER BY [c].[Id]
```

Второй возвращает только счета, но при этом для поиска Id снова происходит выполнение вложенного запроса поиска клиента:

```
SELECT [o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified], [t].[Id]
FROM (
    SELECT TOP(1) [c].[Id]
    FROM [Customers] AS [c]
    WHERE [c].[Email] = N'with@orders.com'
) AS [t]
INNER JOIN [Orders] AS [o] ON [t].[Id] = [o].[CustomerId]
ORDER BY [t].[Id]
```

И вот это поведение, на мой взгляд, не очень эффективное. Если по колонке Email нет индекса, то придется сканировать таблицу Customers дважды. EF должен быть умным и увидеть, что если в первом запросе появился Id клиента, то можно использовать его и во втором. Но по умолчанию снова происходит поиск в виде подзапроса.

Об этом нужно знать, и если нет индекса, то идти на ухищрения. Например, выполнить два запроса и результат объединить в один объект на сервере приложений. Это уже намного эффективнее. Причем такое объединение происходит очень быстро. Но именно тут программисты часто делают ошибки, причем даже опытные. Мне много раз приходилось переписывать код объединения.

Вот пример ужасного ручного маппинга:

```
//bad manual mapping
List<Customer> customers = dbcontext.Customers.ToList();
List<Order> orders = dbcontext.Orders.ToList();
foreach (Customer c in customers)
    c.Orders = orders.Where(m => m.CustomerId == c.Id).ToList();
```

Это плохо, потому что на каждом шаге цикла по клиентам придется сканировать весь список счетов:

```
orders.Where(m => m.CustomerId == c.Id).ToList()
```

На большом списке счетов это будет работать очень медленно.

Более эффективный способ ручного маппинга — создать хеш-таблицу вручную или использовать ToLookup:

```
var ordersLookup = orders.ToLookup(m => m.CustomerId);
foreach (Customer c in customers)
    c.Orders = ordersLookup[c.Id].ToList();
```

Когда мы вызываем orders.ToLookup, то выполняется первый проход через все счета, чтобы построить что-то типа хеш-таблицы. Потом в цикле мы проходим по клиентам, и тут уже обращение к каждому из списков счетов происходит очень быстро.

Что выбрать: ручной маппинг или автоматический с помощью `Include`? По умолчанию можно использовать автоматический. Да, он может загружать лишние данные. Как я уже сказал, если просто вызвать `Include`, то он будет включать все счета, которых может быть очень много. Однако это не проблема, потому что мы можем ограничить выборку, просто указав `Where` прямо в `Include`. Например, в следующем примере я включаю только те счета, которые были созданы до 12 часов 19 декабря:

```
Customer? customer = dbcontext.Customers
    .Where(m => m.Email == "with@orders.com")
    .AsSplitQuery()
    .Include(m => m.Orders.Where(
        m => m.Created > new DateTime(2022,12,19,12,0,0))
    )
    .FirstOrDefault();
```

Таким образом, мы можем с помощью LINQ фильтровать те данные, которые должны подгружаться автоматически. Это только один пример, который пришел в голову, а вы можете использовать всю гибкость LINQ, чтобы построить любые запросы.

Посмотрим, какой запрос был сгенерирован в этом случае:

```
SELECT [t0].[Id], [t0].[Created], [t0].[CustomerId], [t0].[Modified], [t].[Id]
FROM (
    SELECT TOP(1) [c].[Id]
    FROM [Customers] AS [c]
    WHERE [c].[Email] = N'with@orders.com'
) AS [t]
INNER JOIN (
    SELECT [o].[Id], [o].[Created], [o].[CustomerId], [o].[Modified]
    FROM [Orders] AS [o]
    WHERE [o].[Created] > '2022-12-19T12:00:00.0000000'
) AS [t0] ON [t].[Id] = [t0].[CustomerId]
ORDER BY [t].[Id]
```

Читать запрос вам придется не так часто, если работать только с LINQ, но если придется что-то отлаживать и смотреть, где EF работает не так, как вы ожидали, то вот тут можно будет голову сломать.

Ну и напоследок рассмотрим явное включение:

```
using (var dbcontext = new MyStoreContext())
{
    // Явная загрузка
    Customer? customer2 = dbcontext.Customers
        .Where(m => m.Email == "with@orders.com")
        .FirstOrDefault();

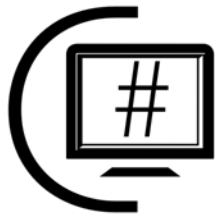
    dbcontext.Entry(customer2).Collection(m => m.Orders).Load();
    foreach (var o in customer2?.Orders)
```

```
{  
    Console.WriteLine($"{o.Id}");  
}  
}
```

Здесь сначала я ищу запись клиента и не использую никаких `Include`, поэтому `customer2.Orders` будет равен `null`. Чтобы подгрузить счета для клиента, который уже загружен, можно обратиться через `Entity` к методу `Collection`:

```
dbcontext.Entry(customer2).Collection(m => m.Orders).Load();
```

Этот код заставит подгрузить счета для клиента, которые не были загружены изначально.



Повторное использование кода

Все проекты, которые мы создавали до сих пор, состояли только из одной сборки. Конечно, когда весь код находится в одном файле, это очень удобно с точки зрения переносимости. Но с точки зрения удобства программирования и повторного использования кода такой подход накладывает серьезные ограничения.

Допустим, при создании определенной программы вы написали очень удобный и функциональный класс. Впоследствии выяснилось, что этот же класс оказался бы весьма полезным в другом приложении. Что делать? Можно включить исполняемые файлы в новый проект и создать новую исполняемую сборку, которая будет содержать необходимые нам классы и методы. Удобно? Нет. Это удобно только при первой компиляции. В реальной жизни такой подход приведет к проблемам, которые мы рассмотрим в *разд. 16.1*.

16.1. Библиотеки

Разделяемый между приложениями код нельзя писать в каждом приложении по отдельности, иначе это приводит к следующим проблемам:

- ❑ если иметь две версии исходных файлов, то при синхронизации изменений файлов первого проекта со вторым могут возникнуть проблемы. Этого можно избежать, если хранить только одну версию исходных файлов и не копировать их в каждый новый проект, но это не избавляет нас от остальных проблем;
- ❑ после внесения изменений в класс приходится перекомпилировать все проекты, в которые включены указанные файлы с исходным кодом. Хорошего решения этой проблемы я не знаю. Можно задействовать утилиты, автоматизирующие компиляцию сразу нескольких проектов, но доставка таких изменений пользователю сделает ваш проект слишком дорогим удовольствием. В случае с веб-программированием можно также использовать микросервисы.

Проблемы решают *библиотеки* кода `Class Library`, которые компилируются в сборки с расширением `dll`. Код из библиотек может загружаться другими сборками и выполняться в домене загружающего приложения. Одна и та же сборка биб-

лиотеки может быть загружена несколькими исполняемыми файлами одновременно, и каждая из них будет выполняться независимо.

Если вам понадобится внести изменения в код библиотеки, которые не затрагивают интерфейсы с внешними сборками, то достаточно лишь перекомпилировать библиотеку и обновить файл сборки библиотеки. Исполняемую сборку перекомпилировать не придется.

Библиотеки кода в .NET обладают одной очень интересной особенностью — они могут быть написаны на любом языке .NET и могут использоваться любым языком этой же платформы без каких-либо ограничений и проблем. На классической платформе Win32 очень часто возникали проблемы совместимости с кодом, написанным на разных языках. Например, языки Delphi и C++ по-разному работают со строками, а также по-разному передают параметры в процедуры, поэтому библиотеки, написанные на Delphi, не так-то легко использовать в приложении на C++.

В .NET все унифицировано. И даже более того. Вы можете создавать потомков от классов, объявленных в библиотеках, и не обращать внимания на язык, на котором был написан предок. Это значит, что класс предка может быть написан на Visual Basic .NET, а потомок — на C# и наоборот. Это очень мощная возможность платформы, потому что теперь программисты, использующие разные языки программирования, могут работать совместно.

В случае с веб-программированием библиотеки позволяют еще больше разделить бизнес-логику и код, который отвечает за генерирование страниц. Только в небольших проектах программисты размещают код работы с базой данных и логику приложения в одном проекте с контроллерами и представлениями. Во всех больших проектах, в которых мне довелось участвовать, бизнес-логика обязательно располагалась в отдельном проекте — библиотеке. А модели представления необходимы именно представлению, и они остаются в веб-проекте.

16.2. Создание библиотеки

Возьмем проект, который мы создавали в *главе 13*, и выделим бизнес-логику в отдельную библиотеку. Для начала нам нужно подправить проект, потому что в нем файл решения MyWebSite.sln и файл проекта MyWebSite.csproj находятся в одной и той же папке. Это плохо и неудобно, если работать больше чем с одним проектом.

Чтобы сделать все правильно, создадим новую папку Project. Скопируем в эту папку только файл MyWebSite.sln. Затем создадим папку Project/Web и в нее скопируем все остальные файлы. Новая структура решения показана на рис. 16.1.

Если сейчас попытаться открыть решение, то произойдет ошибка загрузки проекта, потому что Visual Studio будет искать файл проекта в предыдущем текущем каталоге, а не в подкаталоге Web. Это нужно исправить.

Откройте файл MyWebSite.sln в любом текстовом редакторе. Там у меня на шестой строке прописано подключение проекта к решению:

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "MyWebSite",  
    "MyWebSite.csproj", "{39CA11CD-53DF-4D8C-B3D1-882BC5AE4F0F}"
```

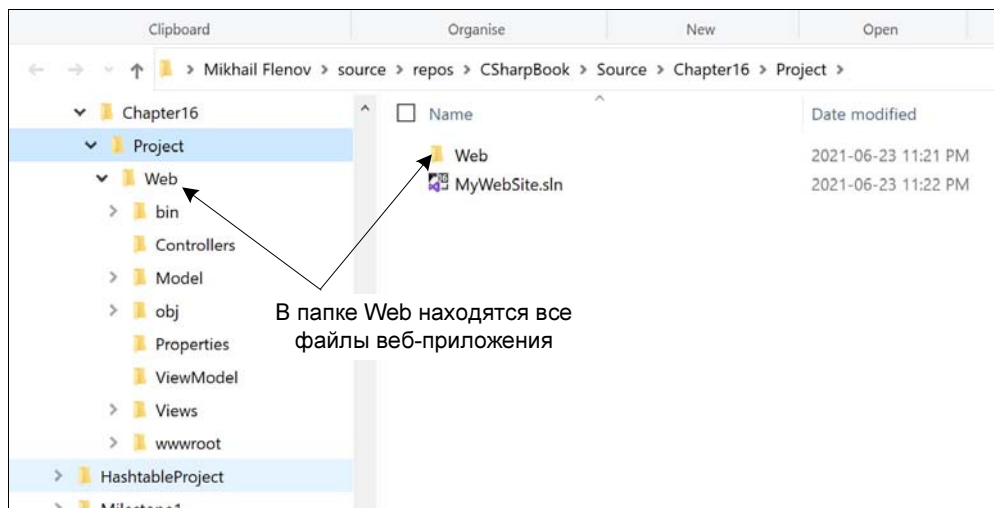


Рис. 16.1. Структура решения

Добавьте перед именем проекта имя папки `web`:

```
Project("{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}") = "MyWebSite",  
    "web/MyWebSite.csproj", "{39CA11CD-53DF-4D8C-B3D1-882BC5AE4F0F}"
```

Вот теперь можно открывать решение, как мы делали это и раньше, и оно должно без проблем открыться.

Перейдем теперь к созданию библиотеки. Для этого щелкните правой кнопкой мыши на имени решения в окне **Solution Explorer** (Проводник решения) и выберите **Add | New Project**. Затем найдите шаблон **Class Library**. Чтобы проще было его отыскать, в третьем выпадающем списке выберите **Library** (рис. 16.2).

На следующем шаге надо выбрать имя библиотеки и ее расположение. По умолчанию расположение должно быть таким же, как и у решения, а в качестве имени библиотеки введем `MyWebSite.Engine`. На последнем шаге выбираем платформу, для которой будет создана библиотека.

После создания проекта шаблон добавит в структуру один файл — `Class1.cs`. Он нам не нужен, и его можно удалить. Это просто пустой файл для кода, в котором разработчики MS предлагают нам начать писать свой код.

Теперь выделите все папки внутри папки `Model` вашего основного приложения (у меня там `DataModel`, `Implementation` и `Interfaces`) и перетащите их в новый проект (рис. 16.3). Это можно сделать в Проводнике решения или в любой программе работы с файлами. Если воспользоваться Проводником решения, то файлы будут скопированы, и у нас образуются две копии одного и того же кода, хотя нам нужна только одна его копия. Так что после копирования папок следует удалить всю папку `Model`.

Если теперь запустить компиляцию, произойдет ошибка, потому что пространство имен `Model` не найдено (рис. 16.4). То есть проект `MyWebSite` не может быть скомпилирован, потому что мы убрали часть кода в отдельную библиотеку.

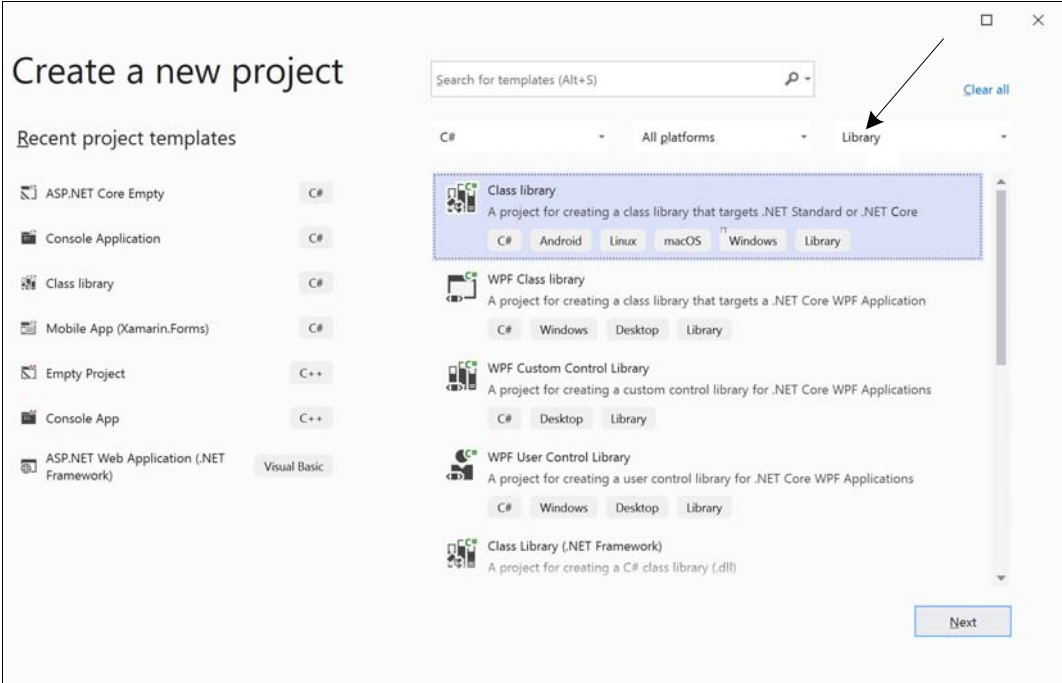


Рис. 16.2. Создание библиотеки

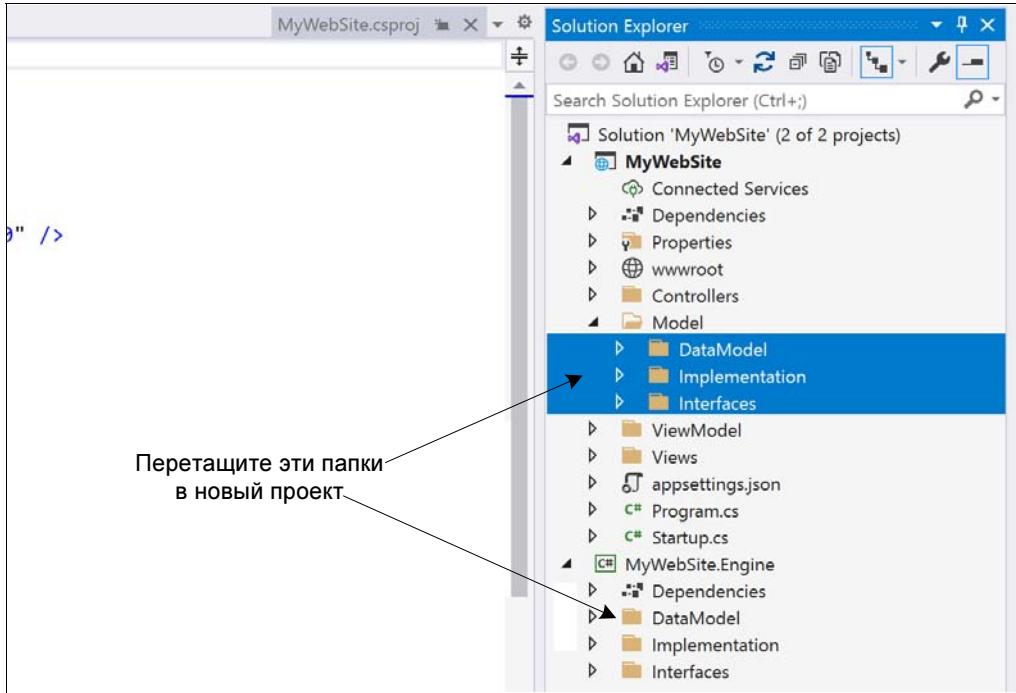


Рис. 16.3. Перетаскиваем файлы из основного проекта в библиотеку

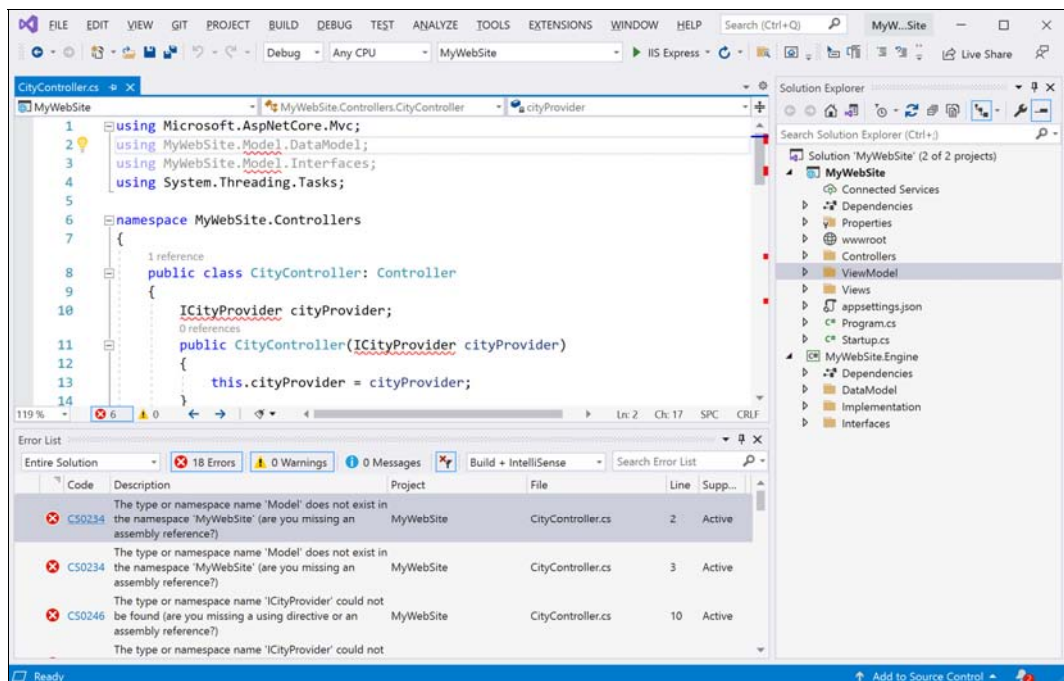


Рис. 16.4. Ошибки компиляции

Если попробовать скомпилировать только библиотеку `MyWebSite.Engine` (щелкнув правой кнопкой мыши на имени проекта **MyWebSite.Engine** и выбрав **Build**), то тоже выявятся ошибки, но здесь чуть понятнее: компилятор не может найти классы библиотеки Dapper для работы с базой данных и конфигурацией. То есть то, как мы в разд. 14.10 подключаем библиотеку Dapper через опцию **Nuget package** к веб-приложению, надо сделать и для нашей библиотеки.

Вы можете сейчас воспользоваться менеджером **Nuget Manager**, чтобы подклю-

- ☐ библиотеку Dapper;
- ☐ пакет `Microsoft.Data.SqlClient`;
- ☐ пакет `Microsoft.Extensions.Configuration.Abstractions`.

Но я покажу вам несколько иной способ. Щелкните на имени **MyWebSite**, и в центральной части окна появится код файла `MyWebSite.csproj`:

```

<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Dapper" Version="2.0.90" />
    <PackageReference Include="Microsoft.Data.SqlClient"
      Version="3.0.0" />
  </ItemGroup>
</Project>

```

```
</ItemGroup>
</Project>
```

Скопируйте из этого кода в память секцию `ItemGroup`, в которой находятся ссылки на пакеты, и можете удалить ее отсюда.

Теперь щелкните на имени проекта **MyWebSite.Engine**, и вы увидите содержимое файла проекта `MyWebSite.Engine.csproj`. Добавьте сюда скопированную секцию `ItemGroup` и добавьте в нее еще одну строку `PackageReference` с указанием на третий пакет, о котором мы упоминали ранее:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Dapper" Version="2.0.90" />
    <PackageReference Include="Microsoft.Data.SqlClient"
      Version="3.0.0" />
    <PackageReference
      Include="Microsoft.Extensions.Configuration.Abstractions"
      Version="5.0.0" />
  </ItemGroup>
</Project>
```

Почему библиотеке нужны три пакета, а в случае с одним веб-проектом нам нужно было два? Дело в том, что проекты используют разную базу. В первой строке исходного кода проекта указан SDK `Microsoft.NET.Sdk.Web`, который в случае с веб-проектом используется по умолчанию, — ведь запускать сайт без конфигурации смысла нет, и, как мы уже видели, она задействуется уже на старте веб-приложения.

В случае с библиотекой базой является `Microsoft.NET.Sdk`, и эта конфигурация может понадобиться, а может быть, и нет, поэтому Microsoft по умолчанию ее не включила, но мы можем добавить ее через ссылку на пакет.

Отлично, наше веб-приложение лишилось кода доступа к базе и всех зависимостей на Dapper.

Теперь если щелкнуть правой кнопкой на **MyWebSite.Engine** в Проводнике решения и выбрать **Build**, то эта операция закончится успешно и в файловой системе появится файл библиотеки `MyWebSite.Engine.dll` — в моем случае он расположен по пути `Source\Chapter16\Project\MyWebSite.Engine\bin\Debug\net5.0\`.

Но вот основной проект мы все еще не можем скомпилировать. Мы убрали зависимости в библиотеку `MyWebSite.Engine.dll`, и они находятся теперь там, но об этом знаем только мы. Так что нужно сообщить и компилятору, где искать эти зависимости.

Щелкните на опции **Dependencies** проекта **MyWebSite** и выберите **Add Project Reference** (рис. 16.5).

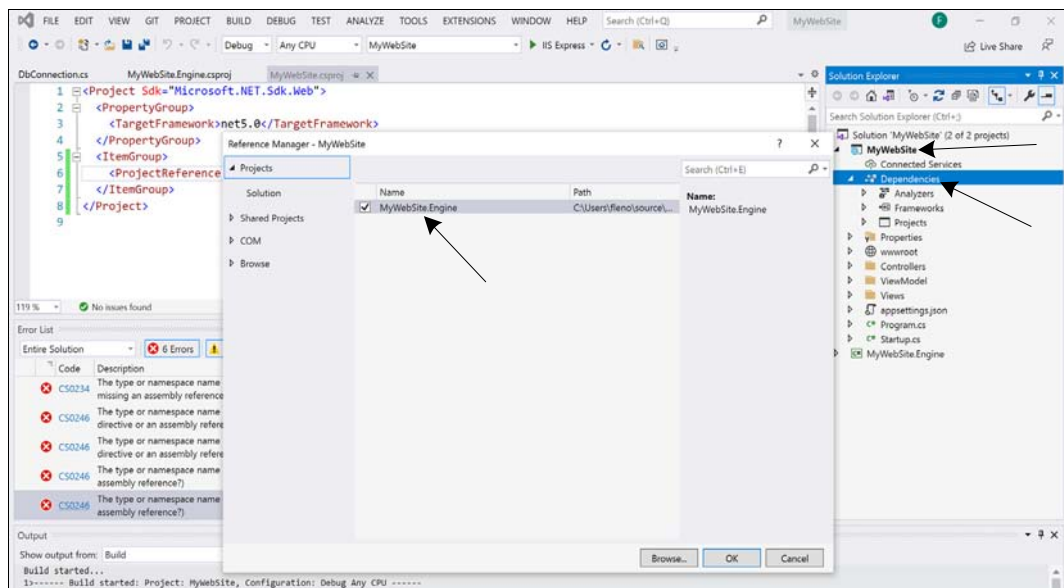


Рис. 16.5. Подключение проекта

В этом окне вы должны будете увидеть все проекты (кроме текущего, потому что подключать проект к самому себе смысла нет) из вашего решения, и вы можете их подключать друг к другу.

С помощью **Nuget** мы можем подключать пакеты, а с помощью **Add Project Reference** — подключать проекты из вашего решения. Идея примерно одна и та же, просто в пакете находятся уже скомпилированные файлы, а при ссылке на проекты вы ссылаетесь как бы на код, и в случае изменения кода он будет перекомпилирован по мере надобности.

Посмотрите на содержимое файла проекта:

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference
      Include="..\MyWebSite.Engine\MyWebSite.Engine.csproj" />
  </ItemGroup>
</Project>
```

Мы снова видим уже знакомый нам `ProjectReference`, в котором есть ссылка на файл `MyWebSite.Engine\MyWebSite.Engine.csproj`.

Вот теперь компиляция должна завершиться успешно, и если запустить сайт, то он должен загрузиться, как и раньше. Но теперь код с логикой находится в совершенно новом проекте.

ПРИМЕЧАНИЕ

Исходный код этого примера можно найти в папке *Source\Chapter16\Project* сопровождающего книгу электронного архива (см. *приложение*).

16.3. Приватные сборки

Созданная нами в *разд. 16.2* библиотека называется *приватной*, потому что она должна находиться в том же каталоге, что и исполняемый файл. За счет того, что мы навели ссылку между проектами, во время компилирования проекта копирование библиотеки в нужную папку происходит автоматически, и нам не нужно об этом заботиться. Если сейчас посмотреть на папку *bin* веб-приложения *\Project\Web\bin\Debug\net5.0*, то вы должны будете увидеть большое количество DLL-файлов, которые мы не создавали (рис. 16.6).

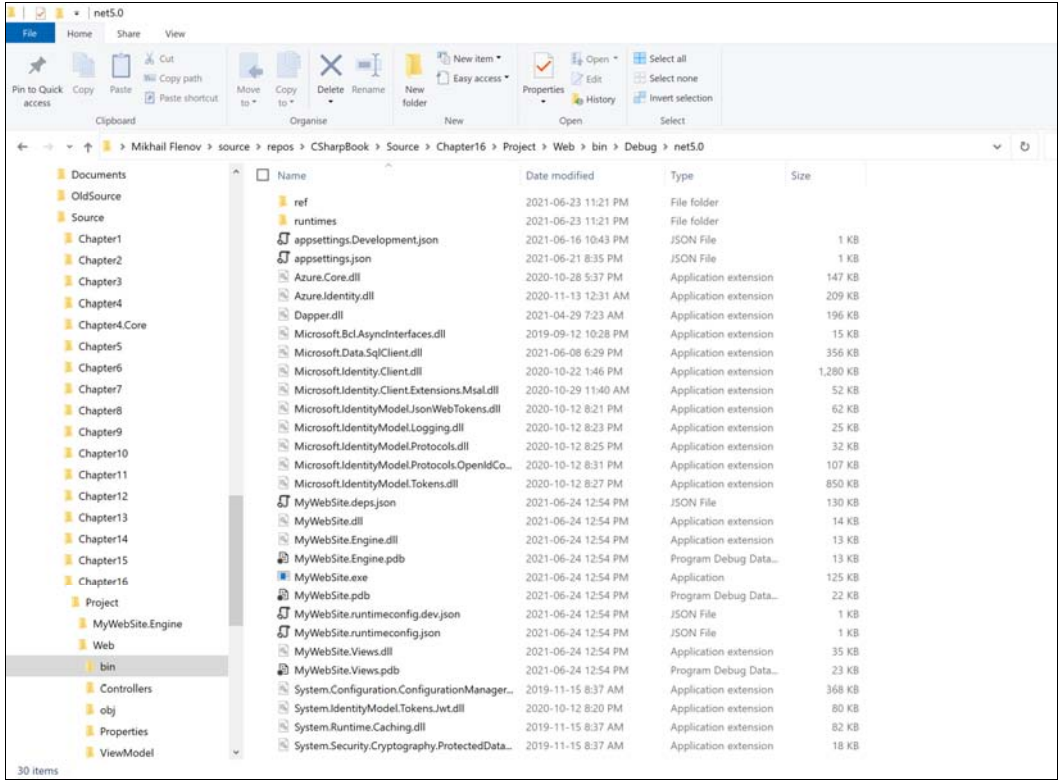


Рис. 16.6. Приватные сборки

Среди них вы найдете и файл *Dapper.dll*, который был создан сообществом, и мы подгрузили его, подключая библиотеку *Dapper*.

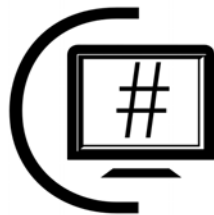
Приватные сборки удобны тем, что их не нужно нигде регистрировать, не нужно как-то устанавливать или, наоборот, деинсталлировать. Вы просто копируете исполняемый файл в определенный (или заранее подготовленный для приложения)

каталог. Для деинсталляции библиотеки достаточно просто удалить ее файл с компьютера или заменить его на более новую версию.

При работе с приватными сборками система не может взять на себя решение проблемы «DLL hell», когда приложение вызывает некорректную версию библиотеки. Решение этой проблемы ложится на плечи разработчика. Это значит, что вы сами должны следить, чтобы исполняемый файл загружал ее корректную версию. Если у вас несколько версий библиотеки, то необходимо контролировать, чтобы программа установки (если вы такой пользуетесь) не перезаписала на компьютере клиента новую библиотеку на более старую.

Когда мы поставляем приложение на сервер, то нужно копировать туда весь этот каталог со всеми DLL-файлами. Вы можете копировать его вручную, а можете воспользоваться специальным мастером, который вызывается по щелчку правой кнопкой мыши на имени проекта и выбору опции **Publish**.

ГЛАВА 17



Сетевые сервисы

Работая над большими проектами, реализовать всю их функциональность в одной большой базе кода сложно и иногда даже невозможно. В таких случаях возникает необходимость коммуникации между разными серверами и приложениями. Обеспечивающие ее технологии менялись несколько раз, так что эту главу пришлось полностью переписать.

В ней мы познакомимся с сетевыми сервисами, а точнее, с их разновидностью — Web API (Application Programming Interface for Web Servers, программным интерфейсом приложения для сетевых серверов).

17.1. История

Прежде чем погружаться в программирование, я хотел бы сделать небольшую паузу и поговорить о теории: когда обращаться к Web API, и почему они сейчас реализуются именно так.

Для начала посмотрим, когда используются сетевые API. Я первый раз столкнулся с необходимостью коммуникации между различными приложениями, когда работал над сайтом rewards.sony.com. На этом сайте пользователи могут заполнить форму на получение кредитной карты Sony, но в реальности карту выдает коммерческий банк. Просто после заполнения формы данные отправляются банку, который их рассматривает и возвращает назад результат рассмотрения, для чего у банка имелся сетевой сервис, с которым я и работал.

Сейчас набирает популярность архитектура микросервисов. Суть этого подхода в том, что функциональность сайта разбивается на некоторое количество небольших сетевых приложений, и когда пользователь загружает сайт, то данные могут собираться в одно целое из разных сервисов.

Мобильные приложения тоже очень часто работают с данными, которые находятся где-то на серверах, и для работы с ними как раз снова отлично подходят сервисы.

Чаще всего сервисы для работе с сайтами реализуются через одностраничные приложения. Ярким примером таких приложений являются сайты для работы с почтой.

Очень часто их создают как раз в виде одностраничных сайтов с богатыми возможностями интерфейса. Когда вы активируете элементы управления на таких сайтах, их страница не перегружается, а изменяется только какая-то небольшая часть сайта. Например, при нажатии кнопки удаления письма может удалиться только выделенная строка письма и параллельно загрузятся новые письма. И вот эти действия как раз выполняются через вызов сервисов на сервере.

Получается, что в коммуникации участвуют две стороны: клиент и сервер. Клиентом может выступать мобильное приложение, другой сервер или сайт, который работает в браузере. Такой клиент отправляет запросы на сервер для выполнения каких-то действий и может получать результат действия или какие-то данные.

Когда еще не было никаких стандартов, каждый программист реализовывал сервисы, как ему хотелось. Но потом пришло время открытых стандартов, и компания IBM начала продвигать сервисы на основе XML. В этом случае клиент оформлял свой запрос в виде XML-файла и направлял его серверу и в ответ также получал XML-документ.

XML-файл может быть достаточно большим, если следовать стандартам. Но даже если упростить XML-запрос до минимума, он все равно может быть достаточно большим:

```
<request>
  <messageid>10</messageid>
  <action>delete</action>
</request>
```

Это простой запрос, который мог бы использоваться, чтобы удалить сообщение с номером 10. Такой запрос обычно отправлялся на порт 80 сервера, поскольку веб-серверы работают именно через этот порт. И это плюс, потому что в больших компаниях доступ в Интернет очень часто лимитирован и защищается сетевыми экранами, но т. к. без сайтов сейчас мало кто может жить, то 80-й порт почти никто блокирует, и таким образом доступ к серверам оказывается свободным.

Недостаток такого подхода — даже маленькое действие требует большого количества текста. И этот текст нужно каждый раз разбирать, чтобы понять запрос. А так как текст большой, то и разбирать приходилось много текста.

Microsoft решила пойти дальше и оптимизировать сервисы, создав свою универсальную технологию, которая бы могла работать как с XML-документами, так и с собственным проприетарным форматом, который был бинарным и поэтому более компактным. Для передачи данных использовался сетевой канал, и вот тут могли возникать проблемы, потому что протокол Microsoft был несовместим с HTTP и приходилось задействовать отдельный порт, который мог быть заблокирован сетевой политикой.

И тут на помощь пришел еще один открытый стандарт, который стал золотой серединой — JSON. Этот текстовый формат более компактный, и он очень часто используется для сериализации объектов, так почему бы не использовать его для сервисов? Такой подход называли Web API, чтобы не путать с Web Service или другими способами коммуникации.

Некоторые считают, что Web API — это те же сервисы, но я все же полагаю, что это немного разные технологии, и поэтому здесь расскажу именно о Web API, называя их Web API или просто API.

Итак, в Web API запросы серверу основаны на классическом HTTP-протоколе. Для доступа к командам можно использовать такие же интернет-адреса (URL), как и в случае с сайтами. Таким образом, команда для удаления письма могла бы выглядеть как обращение к адресу: **`https://www.flenov.ru/message/delete/10`**.

Чтобы отправлять на сервер более сложные запросы, можно использовать POST-запросы, у которых команды могут содержаться в теле HTTP-запроса. Точно так же, как мы отправляли данные на сервер через HTML-формы. Только в теле запроса чаще используют JSON, который компактнее XML:

```
[  
  "Messageid": 10  
]
```

Тут намного меньше символов для передачи по сети, а значит, намного меньше символов придется потом разбирать на сервере, чтобы понять, какая была отправлена команда.

Плюсы такого подхода: компактность, применение популярного и распространенного HTTP-протокола, простота, отсутствие необходимости придумывать новые технологии, возможность использовать существующие серверы, гибкость и т. д.

17.2. Создаем WebAPI

Давайте создадим новый проект, и, как я люблю, будем знакомиться с теорией на практике. В Visual Studio запустите создание нового проекта и выберите **ASP.NET Core Web API**, как показано на рис. 17.1. Остальные параметры оставьте по умолчанию.

Посмотрим теперь на код, который будет в результате сгенерирован и помещен в файл `Program.cs` проекта. Здесь все очень похоже на веб-сайт. Сначала подключаются сервисы:

```
builder.Services.AddControllers();  
builder.Services.AddEndpointsApiExplorer();  
builder.Services.AddSwaggerGen();
```

Первая строка — это подключение контроллеров. Сразу можно обратить внимание на то, что создана папка `Controllers` и в ней сгенерирован файл с контроллером `WeatherForecastController`.

Строка `AddEndpointsApiExplorer` включает поддержку `Api Explorer`, который позволяет находить и описывать аннотации по умолчанию.

Строка `AddSwaggerGen` включает запуск страницы `Swagger`, с помощью которой можно будет вызывать команды нашего API, используя удобный интерфейс. Чтобы `Swagger` заработал, нужно еще его включить, и это происходит в следующих строках:

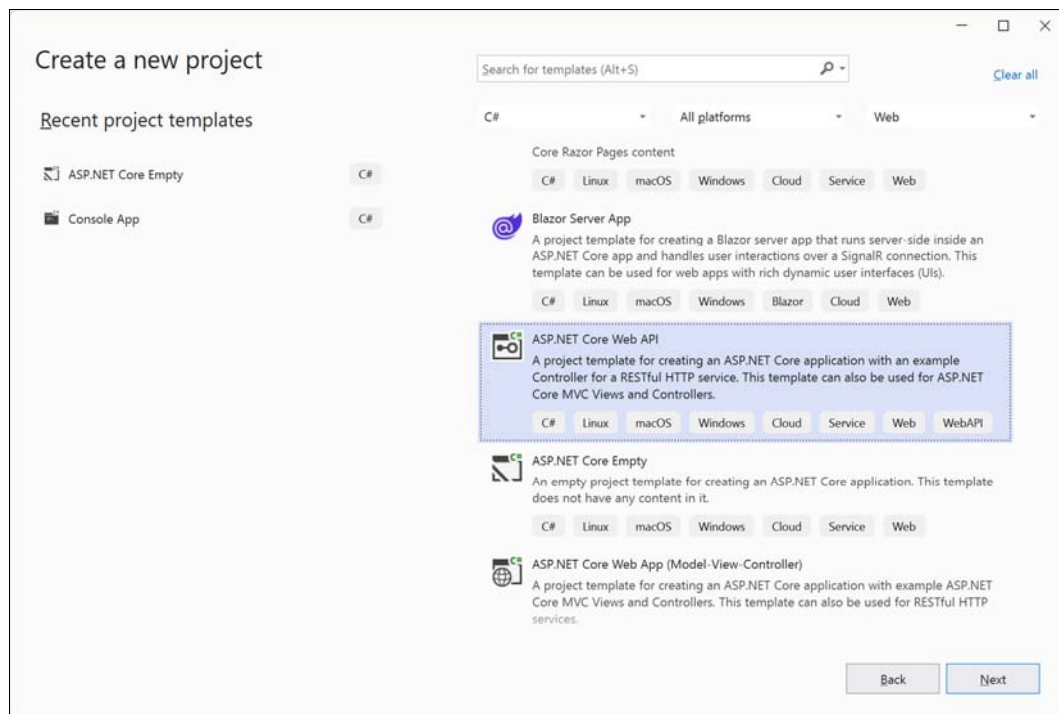


Рис. 17.1. Создание WebAPI проекта

```
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
```

Если мы запустим код в режиме разработки, то Swagger включится. Его и нужно включать только в режиме разработки, а на серверах его лучше не использовать, хотя с точки зрения безопасности отключение Swagger не сильно меняет ситуацию.

Посмотрим на контроллер `WeatherForecastController`. Помимо конструктора, тут только один метод:

```
[HttpGet(Name = "GetWeatherForecast")]
public IEnumerable<WeatherForecast> Get()
{
    return Enumerable.Range(1, 5).Select(index => new WeatherForecast
    {
        Date = DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
        TemperatureC = Random.Shared.Next(-20, 55),
        Summary = Summaries[Random.Shared.Next(Summaries.Length)]
    })
    .ToArray();
}
```

Согласно атрибуту перед методом он будет реагировать на Get-запрос, т. е. вызывать его можно простой загрузкой из браузера. Метод возвращает список из моделей WeatherForecast, и здесь не указан конкретный формат данных, а значит, по умолчанию список объектов будет отформатирован в JSON-формате.

Помните, я говорил, что лучше возвращать объект IActionResult и для работы с JSON для простых сайтов надо использовать JsonResult? Дело в том, что при работе с сайтами нередко приходится делать переадресацию или возвращать вместо данных код ошибки. В случае с API код ошибки тоже можно и даже нужно возвращать в случае проблем, но вот переадресация точно не нужна. Поэтому выгоды от использования IActionResult не будет, и можно возвращать сразу данные.

Настало время запустить наш первый проект. Если запустить его в режиме разработки (Debug), то в окне консоли запустится веб-сервер и тут же загрузится страница Swagger, через которую удобно работать с API.

Если раскрыть строку WeatherForecast, то справа вы увидите кнопку **Try out** (Попробовать), как показано на рис. 17.2. По нажатию этой кнопки чуть ниже появится большая синяя кнопка **Execute**. Нажмите на нее, и произойдет вызов тестового API погоды (рис. 17.3).

Цифра 1 здесь указывает на запрос, который был направлен на сервер, — это URL: **https://localhost:7111/WeatherForecast**. Под цифрой 2 — результат работы API.

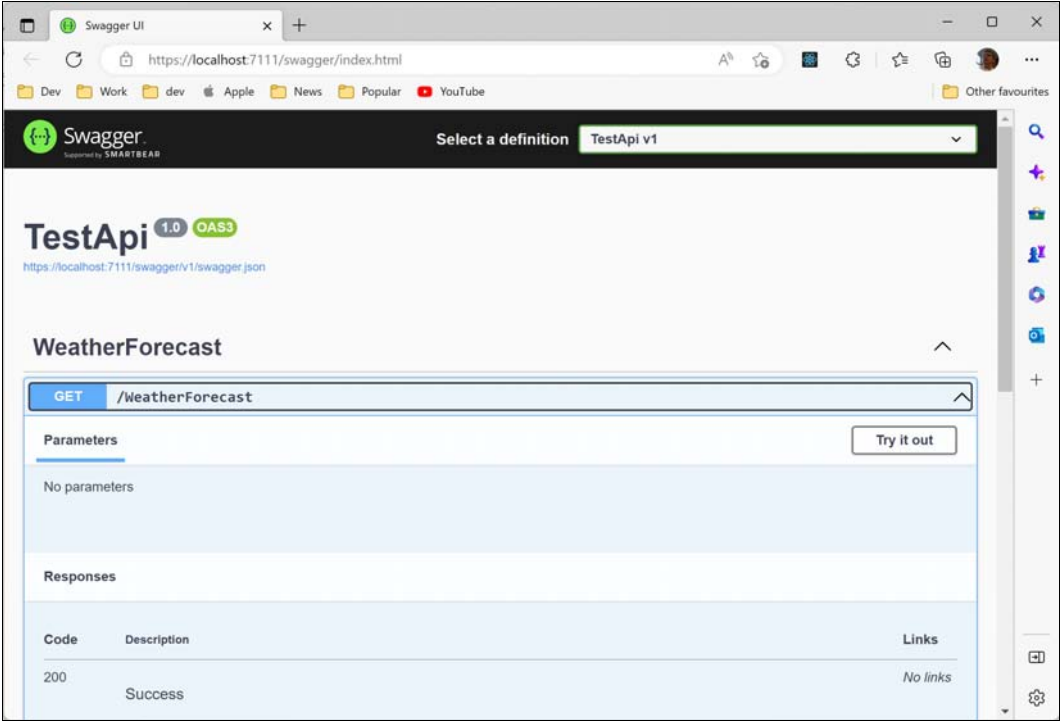


Рис. 17.2. Страница Swagger

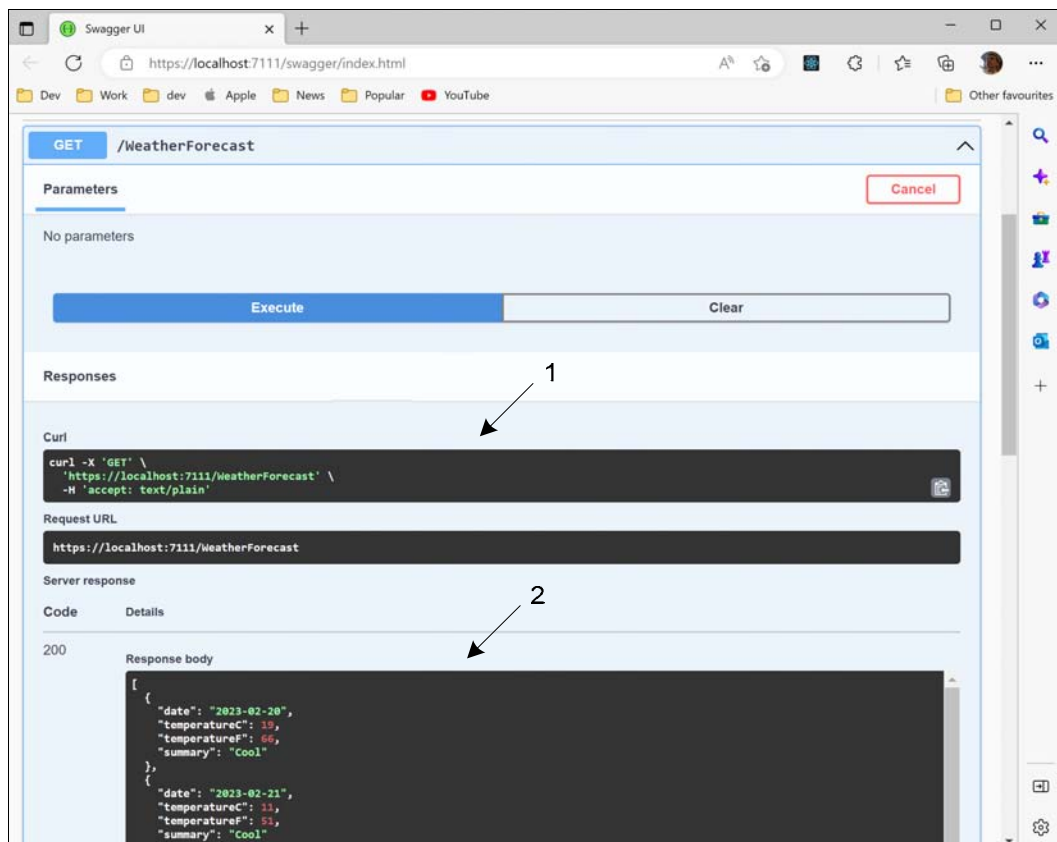


Рис. 17.3. Результат вызова API

Так как наш API вызывается с помощью GET-запросов, никто не мешает нам обратиться к нему из браузера. Попробуйте загрузить URL для API **`https://localhost:7111/WeatherForecast`** в браузере, и вы увидите тот же самый JSON, только без удобного форматирования.

Когда нужно работать с GET-запросами, мы можем использовать браузер, но вся мощь Swagger проявится, когда мы начнем вызывать методы API, которые должны работать с другими методами доступа.

Обратите внимание, что у контроллера есть атрибут: `[ApiController]` и наследует он класс `ControllerBase`. Когда мы создавали сайты, то там контроллеры наследовали класс `Controller`. Дело в том, что при программировании сайтов контроллер может содержать действия для совершенно разных URL. Мы можем использовать тот же подход, что и для построения сайтов: наследовать `Controller` и просто возвращать JSON — это не будет нарушать соглашения. Но Microsoft рекомендует, чтобы мы все же писали свой код немного по-другому: все методы контроллера обрабатывают один и тот же URL, просто каждый метод класса отзывается на свой HTTP-метод. Поэтому атрибут `Route` находится на уровне контроллера, а на уровне методов находится атрибут, в котором указан метод доступа `HttpGet`.

Если и создавать новый метод в этом же контроллере, то он должен обрабатывать другой HTTP-метод, а о доступных методах мы поговорим в *разд. 17.3*.

17.3. Теория Web API

Дизайн хороших Web API в глазах разных программистов выглядит по-разному. Я попытаюсь суммировать эти подходы с моей точки зрения и личного опыта.

Чтобы выполнять какие-то действия на сервере, вы можете создать различные адреса для различных действий, например:

- ❑ `http://localhost:7111/story/add` — для добавления;
- ❑ `http://localhost:7111/story/edit/10` — для редактирования;
- ❑ `http://localhost:7111/story/delete/10` — для удаления.

И можно обращаться к ним с помощью уже знакомых нам методов GET и POST. Я как-то реализовал такие Web API для Sony только потому, что на серверах разрешены были лишь запросы GET и POST, а все остальные были запрещены сетевым экраном.

Дизайн, когда имена действий прошиты прямо в URL, допустим, но лучше использовать более подходящие для этого HTTP-методы.

До сих пор мы работали с двумя методами доступа к методам на сервере: GET и POST, но это не всё — среди стандартных методов есть еще:

- ❑ `HttpGet` или `HttpPost` — получение данных;
- ❑ `HttpDelete` — удаление;
- ❑ `HttpPut` — добавление;
- ❑ `HttpPatch` — обновление.

Получается, что, создавая контроллер для API, мы должны помещать в него методы для работы с какими-то данными с помощью четырех HTTP-методов: получить данные, удалить, добавить, изменить.

Некоторые программисты для добавления используют метод POST, точно так же, как и для простых сайтов. Можно, но я предпочитаю для добавления применять PUT, а GET и POST — для получения данных. Между ними есть большая разница: у GET значения находятся в URL, и их размер ограничен. Тем не менее, работая с API, мне нередко приходилось встречаться с ситуациями, когда запрос содержал большое количество фильтров, и это делало URL слишком большим. В таких случаях намного удобнее и проще воспользоваться POST-запросом на получение данных.

Теперь по поводу ошибок. Если при обращении к Web API произошла ошибка, то некоторые API возвращают в качестве результата код ответа `Http.Ok`, а уже в теле ответа — JSON с примерно таким содержанием:


```
[  
    "Result": "Error",  
    "Description": "Неверный идентификатор"  
]
```

Параметр `Result` здесь излишний — вместо него лучше использовать HTTP-коды.

У нас в контроллере уже есть `HttpGet`-метод, давайте создадим метод `HttpPost`, который не будет выполнять никаких действий, а просто покажет пример отображения ошибки:

```
[HttpPost]  
public string ErrorCodeTest()  
{  
    this.Response.StatusCode = 501;  
    return "Неверный код";  
}
```

Здесь мы устанавливаем свойство `StatusCode` для ответа в значение 501 и возвращаем строку с описанием.

Запустите приложение, и в окне Swagger теперь вы увидите два метода, помеченные разными цветами: `POST`-методы — зеленые, а `GET`-методы — синие (рис. 17.4).

Выполните вариант `POST`, и в результате в окне вы должны увидеть ошибку 501 в качестве кода ответа, а в поле запроса будет текст, описывающий ошибку (рис. 17.5).

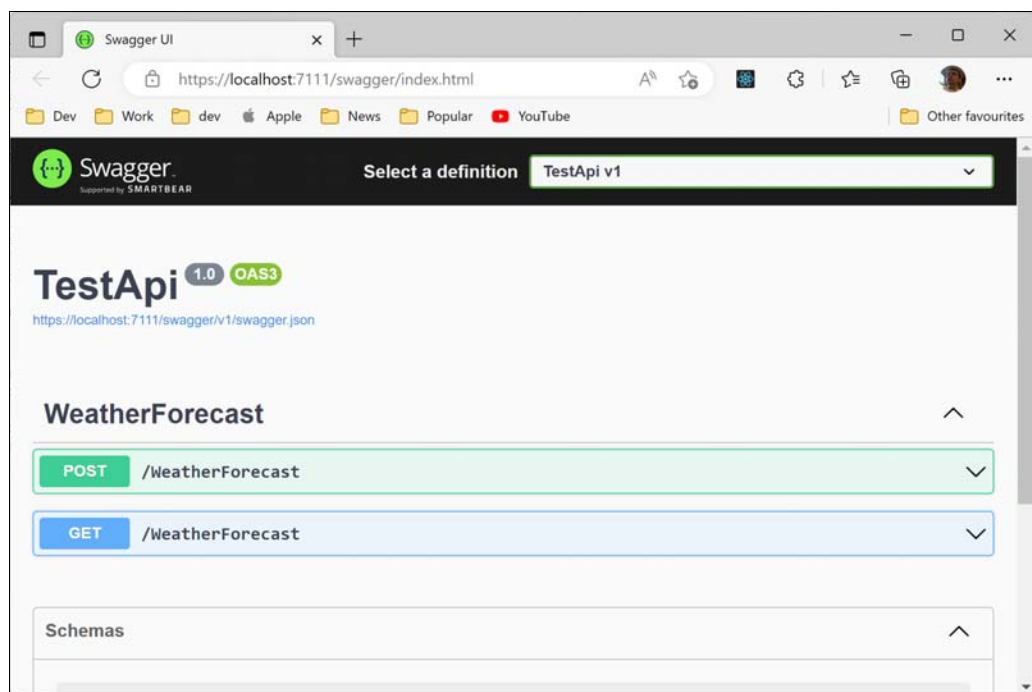


Рис. 17.4. Два тестовых API с разными методами

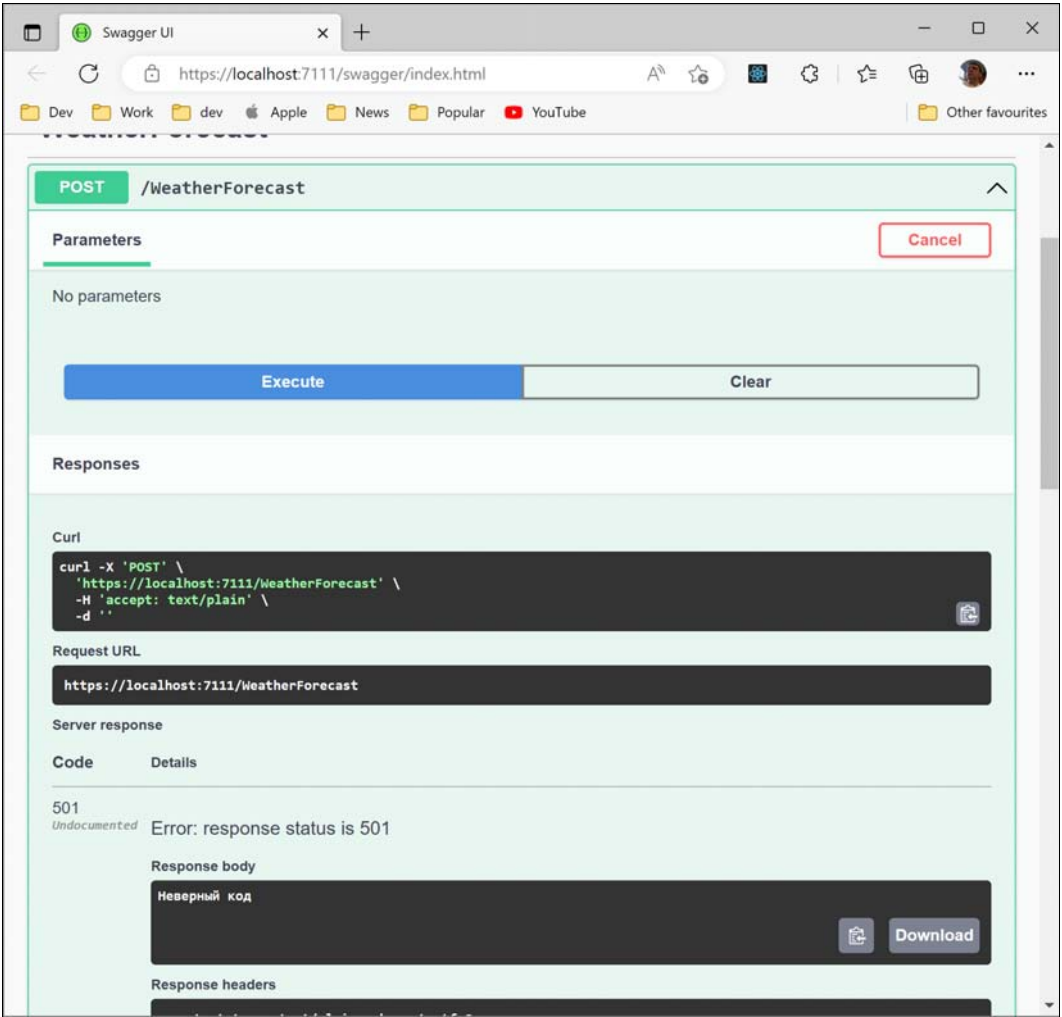


Рис. 17.5. Результат выполнения API с ошибкой

Для каждой ошибки можно заводить свой код результата, и не обязательно возвращать именно 501. Просто в сетевом программировании принято, что если это ошибка, то ее код должен начинаться с 5 и состоять из трех чисел. А описание ошибок вообще возвращать необязательно.

17.4. Сессии

Я уже отмечал, что API могут использоваться мобильными приложениями. А что если мобильному приложению необходимо отслеживать сессию? У API ее нет, но реализовать свою собственную сессию не так уж и сложно.

Я не буду задействовать базу данных, а применю только статичное хранилище в виде статичной переменной, потому что статичные переменные глобальны и су-

ществуют, пока запущен сайт для всех пользователей. Если перезапустить сайт, будет создана новая версия переменной. Так как значения статических переменных разделяются между всеми запросами, в веб-приложениях их нужно использовать аккуратно.

Для хранилища я создал простой класс с двумя методами: получить сессию и создать новую сессию, а данные хранятся в статичной переменной словаря:

```
public class SessionStorage
{
    static Dictionary<string, MemberSession> storage =
        new Dictionary<string, MemberSession>();

    public MemberSession? GetSession(string sessionId)
    {
        if (sessionId != null && storage.ContainsKey(sessionId))
            return storage[sessionId];
        return null;
    }

    public MemberSession CreateSession()
    {
        MemberSession session = new MemberSession();
        session.SessionId = Guid.NewGuid().ToString();
        storage.Add(session.SessionId, session);
        return session;
    }
}
```

Теперь создадим контроллер, который будет использовать эту самостоятельно написанную сессию:

```
[ApiController]
[Route("[controller]")]
public class SessionController : ControllerBase
{
    SessionStorage storage = new SessionStorage();

    [HttpGet]
    public MemberSession Get(string? id)
    {
        var session = storage.GetSession(id);
        if (session == null)
            session = storage.CreateSession();
        return session!;
    }
}
```

У контроллера есть метод `Get`, который возвращает данные сессии по значению идентификатора, полученного в качестве параметра. Если сессия не найдена, то создается новая, и метод возвращает ее.

Теперь во всех своих API в качестве одного из значений нужно передавать значение `id` сессии, которое мы получили при вызове `Get`, — точно так же, как мы передаем значение `Cookie` при работе с сайтами.

В реальном приложении сессия должна быть в базе данных, и желательно, чтобы у нее был срок действия, но для простоты примера в книге этого достаточно.

17.5. Полный цикл

Теперь рассмотрим контроллер, который будет работать с данными в сессии и реализует четыре основных метода: `Get`, `Put`, `Patch` и `Delete`.

Начнем с создания класса для контролера:

```
[ApiController]
[Route("[controller]")]
public class DataController : ControllerBase
```

В этом классе сначала создадим метод `Put`, который будет отвечать за добавление данных:

```
[HttpPost]
public void Put(DataRequest request)
{
    var session = storage.GetSession(request.SessionId);
    if (session == null)
        throw new Exception("Неверная сессия");
    session.Values.Add(request.Name, request.Value);
}
```

В качестве параметра этот метод получает класс `DataRequest`, в котором будут три значения: идентификатор сессии, имя и значение. Если сессия не найдена, то возвращается ошибка.

Запустите приложение, создайте сессию и попробуйте открыть метод `Put`. В результате появится поле для ввода данных запроса в виде JSON (рис. 17.6):

```
{
  "sessionId": "aad29523-df68-4e32-a30d-8d49c4266f22",
  "name": "Test",
  "value": "123"
}
```

Таким образом, мы в виде JSON-строки передаем значение свойств класса `DataRequest`.

Метод `Get` я сделал немного по-другому — здесь значения передаются в виде параметров метода:

```
[HttpGet]
public string Get(string sessionId, string name)
{
    var session = storage.GetSession(sessionId);
```

```
if (session == null)
    throw new Exception("Неверная сессия");
return session.Values[name]?.ToString() ?? "";
}
```

Если запустить теперь сайт и вызвать метод `Get`, то страница Swagger будет выглядеть немного по-другому и просто содержать список значений (рис. 17.7).



Рис. 17.6. Передача данных в виде JSON-запроса

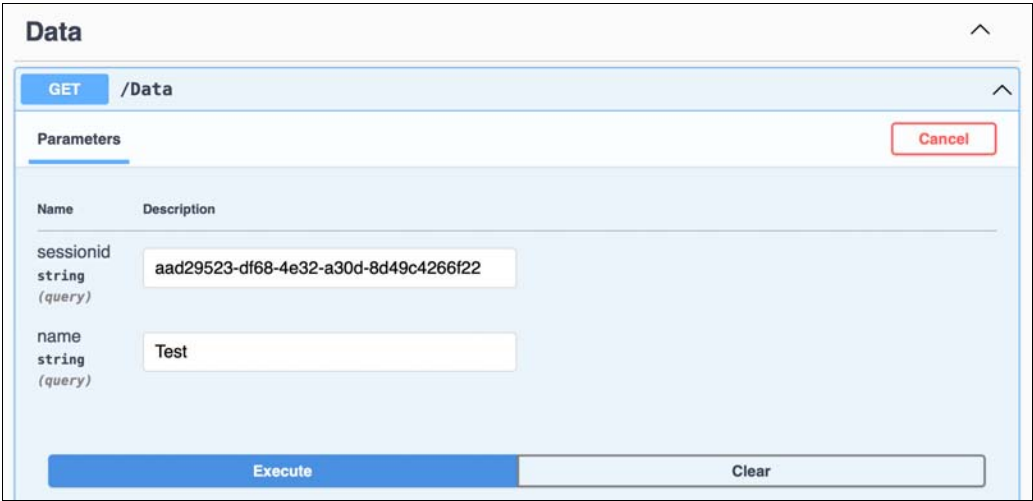


Рис. 17.7. Передача параметров через значения метода

Это не самый эффективный способ оформления контроллера, но я использовал самый простой — ради наглядности.

Методы для изменения и удаления значений работают примерно так же, как и добавления и чтения:

```
[HttpPatch]
public void Patch(DataRequest request)
{
    var session = storage.GetSession(request.SessionId);
    if (session == null)
        throw new Exception("Неверная сессия");
    session.Values[request.Name] = request.Value;
}
```

```
[HttpDelete]
public void Delete(string sessionid, string name)
{
    var session = storage.GetSession(sessionid);
    if (session == null)
        throw new Exception("Неверная сессия");
    session.Values.Remove(name);
}
```

Заключение

В этой книге я постарался заинтересовать вас программированием для платформы .NET с использованием языка C#. Надеюсь, что мне это удалось, и я смог дать вам исходную базу для самостоятельного углубления знаний. Я не пытался повторить официальную документацию или книги других авторов. Моей целью было предоставить вам такие сведения, которые станут основой для использования разнообразных источников информации.

В сопровождающем книгу электронном архиве (см. *приложение*) вы найдете множество дополнительной документации по программированию и не только для .NET. Помимо сведений о программировании, я выложил там достаточно много информации и по Microsoft SQL Server, что будет полезно тем, кто собирается связать свою жизнь с миром баз данных.

Нет ничего идеального, и если вы нашли в книге ошибку, то просьба сообщить мне об этом через мой сайт **www.flenov.info**. Буду рад увидеть любые замечания и комментарии по этой книге.

Список литературы

1. Агуров П. С#. Разработка компонентов в MS Visual Studio 2005/2008. — СПб.: БХВ-Петербург, 2008. — 480 с.
2. Нортроп Т. Разработка защищенных приложений на Visual Basic .NET и Visual C# .NET. Учебный курс Microsoft. — СПб.: БХВ-Петербург, 2007. — 688 с.
3. Фленов М. Transact-SQL. — СПб.: БХВ-Петербург, 2006. — 550 с.
4. Фленов М. Компьютер глазами хакера. — 3-е изд. — СПб.: БХВ-Петербург, 2012. — 272 с.
5. Фленов М. Программирование на C++ глазами хакера. — 2-е изд. — СПб.: БХВ-Петербург, 2009. — 320 с.
6. **www.codeplex.com** — сайт для проектов с открытым исходным кодом от компании Microsoft.
7. **www.codeproject.com** — один из самых старых сайтов по языкам C и C++. Содержит также большой раздел по .NET-технологиям, где можно найти статьи и примеры с исходными кодами по различным темам.
8. **www.flenov.info** — мой сайт, на котором можно найти статьи и заметки о программировании и не только.
9. **www.msdn.com** — классический ресурс любого программиста .NET. Лично я уже не устанавливаю файлы помощи на свой компьютер, потому что они отнимают слишком много места, а все свежее и полезное всегда доступно онлайн в MSDN.

ПРИЛОЖЕНИЕ

Описание электронного архива, сопровождающего книгу

Электронный архив с этой информацией можно скачать по ссылке **<https://zip.bhv.ru/9785977518284.zip>** или со страницы книги на сайте издательства **<https://bhv.ru/>**.

Папки	Описание
\Documents	Дополнительная документация по программированию .NET, по среде разработки Visual Studio и по базам данных
\Source	Исходные коды программ из книги

Предметный указатель

.

.NET 21
.NET Core 20, 31, 32, 35–37, 41
.NET Framework 19, 20, 32, 34, 35, 37

A

as, приведение типов 145

B

base, обращение к предку 138

C

CLR, Common Language Runtime
(Общезыковая среда выполнения) 19, 20
CTS, общая система типов 44

G

GAC, глобальный кеш сборок 37

I

IL-код 21
is, проверка класса 145

J

JIT-компиляция 22

L

LINQ (Language Integrated Query,
интегрированные в язык запросы) 193,
268–278

M

Main(), главный метод 120

P

PDB-файл 30

S

static, ключевое слово 112

T

this, обращение к текущему объекту 109

V

virtual, ключевое слово 134
Visual Studio: переменная среды окружения
34

А

Аксессуары 91

Б

Базы данных 401

◇ транзакции 421

Библиотека Dapper 485

Библиотеки 481

Д

Делегаты 257

Деструктор 118

Домены приложений 320

З

Запросы 268, 269, 271

И

Интерполяция строк 195

Интерфейс 199

◇ IComparable 226

◇ IComparer 227

◇ IEnumerable 222

◇ наследование 208

Инъекция кода 408

Исключения 241

◇ throw 247

◇ блок

▫ catch 245

▫ finally 252

▫ try 244

◇ переполнение 253

К

Класс 87

◇ AppDomain 321

◇ Array 213

◇ ArrayList 218

◇ Console 153

▫ метод Read() 160

▫ метод ReadLine() 160

▫ метод Write() 159

▫ метод WriteLine() 157

◇ Convert 163

◇ DateTime 177

◇ Enum 166

◇ Exception 241

◇ FileStream 285

◇ Hashtable 231

◇ MemoryStream 297

◇ Monitor 317

◇ Object 133

▫ Equals() 135

▫ ToString() 134

◇ OleDbCommand 413

◇ OleDbConnection 403

◇ OleDbTransaction 421

◇ Queue 229

◇ Stack 230

◇ Stream 296

◇ StreamReader 284

◇ StreamWriter 283

◇ String 180

◇ Thread 311

◇ TimeSpan 177

◇ XmlTextWriter 289

◇ абстрактный 147

◇ методы 96

◇ модификаторы доступа 89

◇ псевдоним 122

◇ свойства 90

Комментарии 42

◇ многострочные 43

Компиляция 32

◇ JIT- 22

Константы 79

Конструктор 107

Л

Логическое вычитание 84

Логическое сложение 84

Логическое умножение 84

М

Массивы 55, 213

◇ динамические 217

◇ индексатор 220

◇ невыровненные 215

◇ нумерация 56

◇ типизированные 233

Методы 96

◇ анонимные 267

◇ контроллера 352

- ◇ параметр
 - out 103
 - params 105
 - ref 103
- ◇ перегрузка 107
- ◇ переопределение
 - new 135
 - override 135

О

- Область видимости 142
- Общезыковая среда выполнения (Common Language Runtime, CLR) 19
- Объект 130
- Оператор
 - ◇ break 77
 - ◇ continue 78
 - ◇ if 66
 - ◇ switch 69
 - ◇ typeof 167
 - ◇ using 122
 - ◇ yield 228
 - ◇ перегрузка 183

П

- Переменные 43
- Перечисление 166
 - ◇ ConsoleColor 153
 - ◇ enum 58
- Потоки 310
 - ◇ ввода/вывода 296
- Преобразование типов 162
- Приложение консольное 152
- Приложения MVC 332
- Проект 23
- Пространства имен 122
- Пул потоков 318

Р

- Рефлектор 270
- Решение 23

С

- Сборка 21
 - ◇ версия 38
 - ◇ метаданные 22
 - ◇ приватная 38, 488
 - ◇ разделяемая 38
- Свойства 90
 - ◇ аксессуары 91
- Сериализация 299
- Синтаксис программирования Razor 362
- Словарь 236
- События 258
 - ◇ делегаты 257
 - ◇ синхронные и асинхронные вызовы 266
- Создание представлений 355
- Списки 218
- Структуры 172

Т

- Типы данных
 - ◇ var 146
 - ◇ объектные 44
 - ◇ ссылочные 44

У

- Утилиты
 - ◇ csc.exe 34
 - ◇ ildasm.exe 40

Ц

- Цикл
 - ◇ do..while 75
 - ◇ for 71
 - ◇ foreach 76
 - ◇ while 74

Ш

- Шаблон MVC (Model-View-Controller, Модель-Представление-Контроллер) 343
- Шаблоны 189